

Python 与神经网络实战

何宇健 编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

人工智能已成发展趋势，而深度学习则是其中最有力的工具之一。虽然科技发展速度迅猛，现在实用技术更新换代的频率已经迅速到以周来计算，但是其背后最为基础的知识却是共通的。本书较为全面地介绍了神经网络的诸多基础与进阶的技术，同时还介绍了如何利用神经网络来解决真实世界中的现实任务。本书各章的内容不仅包括经典的传统机器学习算法与神经网络的方方面面，还对它们进行了对比与创新。如果能够掌握本书所讲述的知识，相信即使具体的技术更新得再快，读者也能根据本书所介绍的知识来快速理解、上手与改进它们。

本书兼顾了理论与实践，不仅从公式上推导出神经网络的各种性质，也从实验上对它们进行了验证，比较适合初学者进行学习。同时，本书所给出的框架更能直接、简单、快速地应用在实际任务中，适合相关从业人员使用。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Python 与神经网络实战/何宇健编著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34238-7

I. ①P... II. ①何... III. ①人工神经网络—软件工具—程序设计 IV. ①TP183

中国版本图书馆 CIP 数据核字（2018）第 106127 号

策划编辑：张月萍

责任编辑：刘 舫

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×1092 1/16 印张：25

字数：640 千字

版 次：2018 年 7 月第 1 版

印 次：2018 年 7 月第 1 次印刷

印 数：3500 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

我在写完前一本书——《Python 与机器学习实战》之后，承蒙出版社青睐，被寄予了在某个领域深入剖析并再写一本书的希望。而当时（2017 年年中）恰好是一个非常特殊的时间点——那正是人工智能概念席卷全球，成为当之无愧的“引爆点”的前夕。我在上一本书的前言里曾经说过，自从 AlphaGo 在 2016 年 3 月战胜人类围棋顶尖高手李世石后，“人工智能”“深度学习”这一类词汇就进入了大众的视野；而作为更加宽泛的一个概念——“机器学习”则多少顺势成为从学术界到工业界都相当火热的话题，这也正是我上一本书的主题为机器学习的重要原因之一。而在 2017 年年中时，由于我从方方面面的资料与新闻中都隐隐约约地感受到了深度学习的巨大潜力，所以就与出版社定好了这本书的主题——神经网络。神经网络本身是一个非常宽泛的概念，它既能代指最基础的“全连接神经网络（DNN）”（需要指出的是，DNN 原本泛指 Deep Neural Network，即泛指深层神经网络，不过简洁起见，在本书中我们统一认定它特指全连接神经网络），也能代指当今在各种领域大放异彩的“卷积神经网络（CNN）”和“循环神经网络（RNN）”。本书将主要叙述的是“全连接神经网络（DNN）”及其变体，且其中涉及的技术也能应用在“卷积神经网络（CNN）”和“循环神经网络（RNN）”中。也正因此，在本书的正文中，我们会统一认定“神经网络”代指的是 DNN；但是大家需要知道的是，在前言这里我们会用“神经网络”代指 DNN、CNN 与 RNN 的集合，而在本书以外的场合，虽然说起“神经网络”大家一般都会认为说的是 DNN，但也有可能不单单代指 DNN。

那么，为什么我选择了“神经网络”这个主题呢？简单来说，神经网络是深度学习的“前身”或说“基础”，因为深度学习往简单里说的话，其实就是“比较深的神经网络”。此外，对于本书将主要叙述的 DNN 来说，它和我上一本书中介绍的诸多传统机器学习算法也有千丝万缕的关系；通过两者之间的相互对比，想必大家能对它们都有更深刻的理解，同时也能打下更坚实的基础。

那么，为什么我选择了这种偏基础的主题而不是一些更具体的主题（比如说图像识别、自然语言处理、推荐系统、强化学习）呢？毕竟在有了各种深度学习工具之后，DNN 作为“老前辈”一般的存在，基本只能解决一些结构化数据的问题，而且解决得通常可能还没有传统的机

器学习模型好。这主要出于两个考虑：一是我做的研究本身普遍偏基础，落实到具体的应用时也只是基础方法的具体应用，而非针对具体的应用做的优化；另一方面则是我个人认为，深度学习一旦兴旺起来，各种技术的迭代速度必定是极快的。这是因为深度学习之所以没能获得发展，相当大的原因是受了硬件设备与数据量的制约，一旦这两个制约被打破，那么至少在我看来，它就会以一个月甚至一周的迭代周期来“优胜劣汰”。在这种情况下，即使我花了比较大的力气去介绍我写书时最流行、最有效的技术，等到读者们拿到书时，这些技术很有可能已经没那么流行甚至过时了。总而言之，由于深度学习在我期望中的发展速度过快，所以我没敢向具体的技术下手。

有意思的是，时至今日（2018 年年初），深度学习果然如我所料，几乎可以算是进入了“一周一个新技术”的发展阶段，各大公司的新产品日新月异、层出不穷，这让我为当初选择了基础性的主题而感到庆幸。因为无论具体领域的具体应用技术如何发展，它们背后最本质、最基础的知识都是不会变的。如果能掌握这些基础性的知识，那么在接受各种蜂拥而至的新技术时，想必也会轻松不少吧。

当然，由于我们的目的还是想让读者能够学以致用，所以虽然本书的主题偏基础，但是我们也会花篇幅来介绍如何将这些基础性的知识进行具体的应用。事实上，我们只会在前 5 章介绍理论上的知识，而第 6 章和第 7 章，则都是在介绍如何编写能够应用于现实任务中的框架。此外，本书的代码实现都是偏工程化的，所以大部分核心代码可能会比较长。这是因为我们想要传达的是一种大规模编程下的优良习惯，比如可拓展性、可迁移性、用户友好性等，所以有些地方的实现相比起纯算法实现而言可能会显得略微冗长。不过在我看来，在实际任务中，其实一般很少能够仅仅编写纯算法实现，大部分情况下都需要融合进其他内容，所以相信本书的实现方式能够帮助大家适应今后现实中的情景。

此外需要指出的是，囿于篇幅，本书无法将所有代码悉数放出（事实上这样做的意义也不是很大），所以我们会适当地略去一些相对枯燥且和相应算法的核心思想关系不大、或是测试代码的实现。对于这些代码以及本书展示的所有代码，我都把它们放在了 GitHub 上，大家可以参见 https://github.com/carefree0910/MachineLearning/tree/Book/_Dist/NeuralNetworks 这个目录中的相应代码。我的建议是，在阅读本书之前先把这个链接里面的内容都下载下来作为参照，并在需要的时候通过 Ctrl+F 组合键进行相应的检索。同时，在本书写完之后，由于我会把收笔时的代码保留在 Book 这个 Branch 中，并把后续的更新统一放在 master Branch 中，所以也可以参见 https://github.com/carefree0910/MachineLearning/tree/master/_Dist/NeuralNetworks 这个根目录中的代码以获取无法反映在本书中的、最新的更新。

最后想要说的是，与我写的上一本书类似，虽然本书会尽量避免罗列枯燥的数学公式，但是一些比较重要的公式与证明还是不可或缺的。不过考虑到数学基础因人而异，我把一些额外的、类似于“附加证明”的章节打上星号（*）。对于之前学过机器学习、基础比较扎实的读者，阅读这些带星号的章节是比较有益的，因为它们有助于更深刻地理解一些知识背后的理论；而对于零基础或仅想通过本书入门的读者，这些章节可能稍显困难，所以直接把它们跳过也是不错的选择，因为跳过它们不看并不会对理解主要内容造成很大的影响。

本书特点

- **注重基础**：本书不仅介绍了神经网络的基础知识（比如前向传导算法、反向传播算法、Dropout、Batch Normalization 等），还涵盖了传统的、经典的、基础的机器学习算法（朴素贝叶斯、决策树、支持向量机）的大意以及这些算法与神经网络在本质上的联系。
- **注重创新**：本书将会介绍许多属于新颖的、有效的技术。比如第 4 章中的转换算法，目前市面上的图书中基本没有类似的、面面俱到的介绍；而第 5 章介绍的神经网络中的软剪枝技术，则更是笔者个人的研究，可以说是“只此一家”。
- **注重应用**：本书最后两章将会着重介绍如何搭建在现实任务中确实能拿来应用的神经网络框架，该框架不仅兼顾了可扩展性，也兼顾了用户友好性，能在各个领域中发挥作用。
- **注重基础**：本书中的所有代码实现沿袭了上一本书的传统，基本都是“从零开始”。也正因此，所涉及的核心实现大多仅仅基于一些基础的第三方库（如 `numpy`、`TensorFlow`）而没有依赖更高级的第三方库，这使得我们能够通过代码实现来辅助理解算法细节。

本书的内容安排

第 1 章 绪论

本章介绍了一些基本概念与基础术语，这些内容虽然可能略显乏味，但却是跨入业界必须掌握的知识。本章的篇幅很短，内容相对浓缩，对于零基础的读者可能会需要花一些时间去消化，对于有基础的读者则是比较好的知识唤醒。

第 2 章 经典传统机器学习算法简介

本章将会精要介绍 5 个经典的传统机器学习算法——朴素贝叶斯、决策树、感知机、支持向量机和 Logistic 回归，介绍的方式偏向于直观叙述，旨在让大家更好地从原理而不是从细节上去理解这些算法。虽然本书的主题是（全连接）神经网络，但是了解这些传统的机器学习算法是有必要的，它们不仅能给我们解决实际问题的思维，也能佐证神经网络的强大。

第 3 章 神经网络入门

本章是神经网络的入门章节，旨在全面且较为深刻地叙述神经网络的各项技术要点，包括神经网络的结构、激活函数、损失函数、前向传导算法、反向传播算法、梯度下降法的各式变体、搭建 TensorFlow 模型基本框架的诸多注意事项等。如果能够将本章的所有内容都深刻理解的话，相信不仅能对神经网络有比较扎实的认知，也能对 TensorFlow 这个深度学习框架有更好的认知，这对今后的各种实际应用是大有裨益的。

第 4 章 从传统算法走向神经网络

本章通过将神经网络与第 2 章所介绍的各式传统机器学习算法做对比，以此来帮助大家加

深对这些传统机器学习算法和神经网络的理解。我们将会看到，即使是最朴素的神经网络，就已经能够自然地表达出感知机、支持向量机和 Logistic 回归这三种模型，而且只要通过定制它的权值矩阵与偏置量，甚至还能够表达出几乎任意的朴素贝叶斯模型与决策树模型。此外，这种表达还有着一定的现实意义，并不仅仅是理论上的技巧。

第 5 章 神经网络进阶

本章是神经网络的进阶章节，旨在介绍一些能够提升朴素神经网络性能的技术，包括 Dropout、Batch Normalization、Wide and Deep、Deep Neural Decision Forest、Dynamic Network Surgery 以及软剪枝技术等。这些技术并不是一个个独立的技术，在经过适当整合之后，从理论和实验效果来看都会拥有许多有趣且有用的性质。

第 6 章 半自动化机器学习框架

本章旨在介绍如何将前 5 章介绍过的技术应用到实际问题中。虽然在介绍技术理论时我们可以做出许多很强的假设，比如数据都是连续型数据、各个特征的取值都很合理，以及训练步数可以控制在最优等，但对于真实世界中的数据集而言，我们往往需要做很多预处理工作之后，才能够应用诸多（神经网络的）技术；而且由梯度下降法的特性可知，训练步数是一个理论上不可知的、但是又非常重要的超参数，究竟何时停止训练甚至是模型表现优秀与否的最关键的环节。本章会较为全面地给出针对这些问题的一套解决方案，这套解决方案能够直接应用于绝大多数非结构化数据上，实现了机器学习的“半自动化”。

第 7 章 工程化机器学习框架

前 6 章的内容已经使我们能够用同一个模型端到端地解决许多问题，但是在现实任务中，我们不仅常常需要进行多次重复实验来更好地验证模型，而且还常常需要同时对很多模型做实验并选出其中最好的模型，这就要求我们在前 6 章的基础上进一步自动化这些功能。本章将会介绍如何利用前 6 章实现的框架来进行多次实验与参数搜索，前者可以更合理地评估特定模型在特定数据集上的性能，而后者则能够挑选出最适合特定数据集的模型。

适合阅读本书的读者

- 想要转行 AI、投身 AI 的程序员
- 想要知道 TensorFlow 编程技巧的程序员
- 想要打好基础并入门深度学习的学生、老师、在职员工等
- 想要使用神经网络来解决（非结构化数据分类预测的）实际问题的从业者

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 扫码直达本书页面。

- **下载资源**: 本书中部分图片的彩色版本可在[下载资源处](#)下载。
- **提交勘误**: 您对书中内容的修改意见可在[提交勘误处](#)提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- **交流互动**: 在页面下方[读者评论处](#)留下您的疑问或观点, 与我们和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/34238>



目 录

第 1 章 绪论	1
1.1 机器学习简介	2
1.1.1 什么是机器学习	2
1.1.2 机器学习常用术语	3
1.2 Python 简介	9
1.2.1 Python 的优势	10
1.2.2 scikit-learn 和 TensorFlow	11
1.3 前期准备	13
1.3.1 训练、交叉验证与测试	13
1.3.2 简易数据预处理	14
1.4 本章小结	15
第 2 章 经典传统机器学习算法简介	17
2.1 朴素贝叶斯	17
2.1.1 条件独立性假设	18
2.1.2 贝叶斯思维	19
2.1.3 模型算法	20
2.1.4 实例演示	23
2.1.5* 参数估计	25
2.1.6* 朴素贝叶斯的改进	28
2.2 决策树	33
2.2.1 决策的方法	33

2.2.2	决策树的生成	34
2.2.3	决策树的剪枝	39
2.2.4	实例演示	40
2.2.5*	决策树的三大算法	40
2.2.6*	数据集的划分	45
2.2.7*	决策树与回归	48
2.3	支持向量机	50
2.3.1	分离超平面与几何间隔	50
2.3.2*	感知机与 SVM 的原始形式	58
2.3.3	梯度下降法	62
2.3.4*	核技巧	70
2.3.5	实例演示	75
2.4	Logistic 回归	75
2.5	本章小结	76
第 3 章	神经网络入门	77
3.1	神经网络的结构	78
3.2	前向传导算法	80
3.2.1	算法概述	81
3.2.2	算法内涵	83
3.2.3	激活函数	85
3.2.4	损失函数	90
3.3*	反向传播算法	92
3.3.1	算法概述	92
3.3.2	损失函数的选择	94
3.4	参数的更新	98
3.4.1	Vanilla Update	99
3.4.2	Momentum Update	99
3.4.3	Nesterov Momentum Update	100
3.4.4	AdaGrad	100
3.4.5	RMSProp	101
3.4.6	Adam	101
3.5	TensorFlow 模型的基本框架	101
3.5.1	TensorFlow 的组成单元与基本思想	102
3.5.2	TensorFlow 模型的基本元素	104
3.5.3	TensorFlow 元素的整合方法	114

3.5.4 TensorFlow 模型的 save & load	125
3.6 朴素神经网络的实现与评估	130
3.7 本章小结	138
第 4 章 从传统算法走向神经网络	139
4.1 朴素贝叶斯的线性形式	139
4.2 决策树生成算法的本质	145
4.2.1 第 1 隐藏层→决策超平面	147
4.2.2 第 2 隐藏层→决策路径	148
4.2.3 输出层→叶节点	150
4.2.4 具体实现	151
4.3 模型转换的实际意义	158
4.3.1 利用 Softmax 来赋予概率意义	159
4.3.2 利用 Tanh+Softmax 来“软化”模型	160
4.3.3 通过微调来缓解“条件独立性假设”	165
4.3.4 通过微调来丰富超平面的选择	165
4.3.5 模型逆转换的可能性	171
4.4 模型转换的局限性	172
4.5 本章小结	172
第 5 章 神经网络进阶	174
5.1 层结构内部的额外工作	175
5.1.1 Dropout	175
5.1.2 Batch Normalization	176
5.1.3 具体实现	180
5.2 “浅”与“深”的结合	181
5.2.1 离散型特征的处理方式	181
5.2.2 Wide and Deep 模型概述	183
5.2.3 Wide and Deep 的具体实现	185
5.2.4 WnD 的重要思想与优缺点	194
5.3 神经网络中的“决策树”	195
5.3.1 DNDF 结构概述	195
5.3.2* DNDF 的具体实现	199
5.3.3 DNDF 的应用场景	210
5.3.4* DNDF 的结构内涵	213

5.4	神经网络中的剪枝	216
5.4.1	Surgery 算法概述	216
5.4.2	Surgery 算法改进	218
5.4.3	软剪枝的具体实现	221
5.4.4*	软剪枝的算法内涵	223
5.5	AdvancedNN 的结构设计	237
5.5.1	AdvancedNN 的实现补足	237
5.5.2	WnD 与 DNDF	239
5.5.3	DNDF 与剪枝	241
5.5.4	剪枝与 Dropout	242
5.5.5	没有免费的午餐	242
5.6	AdvancedNN 的实际性能	243
5.7	本章小结	251
第 6 章	半自动化机器学习框架	253
6.1	数据的准备	254
6.1.1	数据预处理的流程	254
6.1.2	数据准备的流程	256
6.2	数据的转换	264
6.2.1	数据的数值化	264
6.2.2	冗余特征的去除	266
6.2.3	缺失值处理	269
6.2.4	连续型特征的数据预处理	272
6.2.5	特殊类型数据的处理	274
6.3	AutoBase 的实现补足	277
6.4	AutoMeta 的实现	281
6.5	训练过程的监控	288
6.5.1	监控训练过程的原理	288
6.5.2	监控训练的实现思路	292
6.5.3	监控训练的具体代码	293
6.6	本章小结	299
第 7 章	工程化机器学习框架	301
7.1	输出信息的管理	301
7.2	多次实验的管理	309

7.2.1	多次实验的框架	312
7.2.2	多次实验的初始化	314
7.2.3	多次实验中的数据划分	316
7.2.4	多次实验中的模型评估	318
7.2.5	多次实验的收尾工作	321
7.3	参数搜索的管理	321
7.3.1	参数搜索的框架	322
7.3.2*	随机搜索与网格搜索	329
7.3.3	参数的选取	334
7.3.4	参数搜索的收尾工作	335
7.3.5	具体的搜索方案	335
7.4	DistAdvanced 的性能	337
7.5	本章小结	344
附录 A	SVM 的 TensorFlow 实现	345
附录 B	numba 的基本应用	352
附录 C	装饰器的基本应用	359
附录 D	可视化	363
附录 E	模型的评估指标	370
附录 F	实现补足	377

第 1 章

绪论

第一次工业革命约兴起于 17 世纪 60 年代，一直持续到 18 世纪 30 年代至 18 世纪 40 年代，人类在这段时间里使用机器取代人力、兽力，以大规模的工厂生产取代个体手工生产。第二次工业革命则紧跟着第一次工业革命之后，以电力的大规模应用为代表，新技术的重要性可谓展现得淋漓尽致。第三次工业革命常被称为科技革命、信息技术革命、数字化革命，计算机和电子数据正是在这期间普及、推广的，使得各行各业都发生了从机械和模拟电路到数字电路的变革，至今仍未结束。如果稍微总结一下的话，大抵可以这样说：第一次工业革命由机器驱动，第二次工业革命由电力驱动，第三次工业革命则由数字化与新科技驱动。

而第四次工业革命，不少人认为将会是集前三次工业革命之大成。具体一点说，不少人认为它将由人工智能驱动。

人工智能（Artificial Intelligence，常简称为 AI）在近期已经有人尽皆知的趋势，不少行业的领军企业都纷纷提出了“AI First”的策略。人工智能的魅力毋庸置疑，从小说到电影、从纪实到科幻，人工智能永远是一个热门的题材。

这也使得许多人对人工智能心生向往，但同时又可能不知如何下手。本书的目的就是为广大读者提供其中一个入手的方向——使用 Python 来搭建神经网络模型。须知神经网络乃深度学习的根基，而深度学习则是人工智能中最有名、有效的工具之一。所以在学习了神经网络相应的理论后，相信大家就能以此为基础进而开发出人工智能的应用，从而为这个 AI 社会贡献出自己的一份力量。作为本书的第 1 章，我们打算先谈谈入门 Python 神经网络所需的一些比较宽泛的知识。我们会在第 1 节进行与机器学习相关的一些说明，在第 2 节、第 3 节介绍一下 Python 和一些 Python 中常用的第三方库和小工具。

具体而言，本章主要涉及的知识点有：

- 机器学习的定义及术语

- Python 在机器学习领域的优异性
- scikit-learn 和 TensorFlow 的简介与安装
- 数据的简易预处理与进度条、计时器的使用

1.1 机器学习简介

由于人工智能在最近才火起来（不如说在以前，人们通常并不以做人工智能为荣，反而羞于说自己是做人工智能的），所以关于它的一些概念存在着一定的不统一性。本节打算对这些概念进行梳理，以方便本书后文的讨论。

首先大家可能想问：机器学习(Machine Learning, 常简称为 ML)、神经网络(Neural Network, 常简称为 NN)、深度学习(Deep Learning, 常简称为 DL)和人工智能(AI)之间的关系到底是什么？这个问题的答案可以归结为如下三点：

- 神经网络是深度学习的“根基”。
- 深度学习是机器学习的一个分支。
- 机器学习是人工智能领域的一种技术。

换句话说，我们可以使用机器学习技术来做人工智能，而深度学习则是众多机器学习技术中效果最好的技术之一（围棋界的 Master 是最具代表性的存在；当然，Master 所用的技术并不全是深度学习，但核心技术大都是深度学习）。深度学习比较严谨的定义可以在维基百科（Wikipedia）上找到——它是试图使用包含复杂结构或由多重非线性变换构成的多个处理层对数据进行高层抽象的、基于对数据进行表征学习的一种算法。不过通俗地说，它其实就是各种花式神经网络的集合。事实上维基百科中也说了，“深度学习”已成为类似术语，或者说是神经网络的品牌重塑。

神经网络单单凭借其名字及效果，其实就已经使不少人或多或少地对它产生了兴趣与敬畏。其看上去很复杂的数学公式，更是使得许多初学者望而却步。但是如果我们静下心来一步步踏实地走下来，就会发现神经网络其实并没有想象中那么可怕。为了更好地理解神经网络，对传统机器学习算法进行介绍（第 2 章）并用神经网络与它们进行对比（第 4 章）是有必要的，为此我们需要先知道机器学习的基本思想和其中一些常见的术语。

1.1.1 什么是机器学习

机器学习听上去非常玄妙，但它本质上没有多么复杂。要理解这个词语，我们一般会把它拆成“机器”与“学习”这两部分来理解。其中，“机器”是拥有“没有生命”“能够进行运算”“服从人类指令”等基本属性的物件，比如，我们的电脑或服务器等硬件设备。而所谓的“学习”，和我们人类所说的学习有相近之处，但却又不完全一致。人类在学习时，常常是通过五感（视觉、听觉、触觉、嗅觉和味觉）与外界进行交互来接收信息，然后结合过往的经验来理解、吸收。而且我们在年龄尚小、自我监督能力不够强时，常常还需要一定的激励才能很好地将学

习继续下去（比如考试拿到一定的分数就能获得想要的礼物之类的）。而对于机器来说，它与外界的交互可以说只有一个媒介——那就是数据。同时，虽然我们有时需要设计一种包含激励的算法（比如说强化学习），但机器不会因为不具有激励就不进行学习。只要我们将学习的指令下达给机器，机器就会不知疲倦地执行相应的步骤。

所以对于机器学习而言，核心的内容其实只有两个：算法（赋予“学习能力”的指令）和数据（“学习”的素材）。在传统的机器学习模型中，算法往往是比较关键的部分；数据虽然也至关重要，但我们往往会着重要求它的“质量”而非“数量”。不过对于本书将主要讨论的算法——神经网络而言，数据的“数量”往往是最为重要的一环（当然，质量也很重要），这也是为什么不少人称之为“数据驱动模型”。具体的细节我们会在第3章第6节的最后给出一个解释，此处就暂时按下不表。

值得一提的是，在笔者写下这段话时，刚出了一个很厉害的东西，叫作“AlphaGo Zero”，它是没有利用任何数据、从头学习的围棋 AI，但其最终性能却比以往所有利用了数据的围棋 AI 都要强。然而笔者认为，AlphaGo Zero 的成功虽然确实说明了数据在特定领域并不是必需的，但却并不能说明数据的重要性降低了。事实上我们在讨论围棋时，常常以其“规则简单却博大精深”来赞扬它，而这个“规则简单”很可能恰恰是 AlphaGo Zero 无须人工数据的关键。引用周志华教授的看法之一就是：“别幻想什么无监督学习，监督信息来自精准规则，非常强的监督信息”。当我们面对一个实际问题时，规则常常是模糊且充满不确定性的（比如股票交易问题），此时大量的数据是不可或缺的。

1.1.2 机器学习常用术语

正如物理学中有势能、动能、惯性、功率等专有名词，化学中有元素周期表作为根基，数学更是有完备的符号体系来进行表述，机器学习领域也有一套术语方便同行间的沟通、交流与讨论。这些术语乍一听可能会觉得非常“高大上”——“样本空间”“交叉验证集”“假设空间”“泛化能力”……然而我们需要知道的是，这些术语实际上都是可以用非常直观的方式来理解的。虽然它们背后的数学背景可能会五花八门（比如概率论、数理统计、实变与泛函等），但是即使我们撇开过于细节的知识，也不会对实际应用乃至研发和创新带来很大的影响（当然，如果真的能做到顶尖的话，相应的数学背景还是要补上的，不过那就超出本书的讨论范围了）。

我们在 1.1.1 节刚说过，神经网络常常被称为“数据驱动模型”，下面就来看看和模型、数据相关的一些术语和相应的默认符号。它们都是非常基础且重要的，是需要被牢牢记住并掌握的。

- 模型（Model），它是某个机器学习算法所导出的、能够完成训练与预测等任务的物件，我们常常用 G 来指代它。我们的任务就是使用数据来训练 G ，并让 G 能够为我们解决一系列特定的事情。而模型本身又可以衍生出两个相关的概念：
 - 参数（Parameter），它是决定模型行为的东西，我们的训练目的就是把参数训练到一个能够使模型表现最好的值。一般而言，参数会用 θ 来代指，从而我们的模型

G 常常可以写成：

$$G(\mathbf{x}) = G(\mathbf{x}|\theta)$$

从直观上来说， $G(\mathbf{x}|\theta)$ 表示的就是“ G 在参数 θ 下的行为”。

- 。超参数（Hyper Parameter），它是决定模型结构或训练行为的东西。与参数不同的是，超参数一般是不能被训练而只能被“选择”的。换句话说，我们可以通过选择不同的超参数来搭建出同一套算法下结构和训练方式不一的模型，但我们在训练的过程中，超参数常常是保持不变的。超参数可以用 $\tilde{\theta}$ 来代指，不过需要单独指明超参数的场景很少，我们一般会将超参数和参数视为一个整体，并把这个整体用 θ （大写的 θ ）来表示，从而我们的模型 G 常常可以写成：

$$G(\mathbf{x}) = G(\mathbf{x}|\theta)$$

从直观上来说， $G(\mathbf{x}|\theta)$ 表示的就是“ G 在参数和超参数的集合 θ 下的行为”。如果没有进行特殊说明，后文中我们会直接称 θ 为参数，而不一一仔细区分参数与超参数。

- 空间（Space），它常作为后缀出现，如果没有特别指出，那么它表示的就是前面的词语“可能存在的取值”。比如，我们说“模型空间”时，指的就是“所有可能的模型”；当我们说“参数空间”时，指的就是“所有可能的参数”；当我们说“样本空间”（常用 \mathcal{D} 代指）时，指的就是“所有可能的样本”（样本的定义见下）。
- 数据集（Data Set），顾名思义，它是数据的集合。
- 样本（Sample），它是数据集中的每一条单独的数据。如果没有特殊说明，我们会默认数据集中有 N 个样本，并用符号：

$$D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$$

来表示数据集。不难看出，其中的

$$(\mathbf{x}_i, y_i), \quad i = 1, 2, \dots, N$$

就是一个样本，它包含以下两个部分。

- 。特征向量（Feature Vector），它是样本中的 \mathbf{x}_i ，常常是模型输入的来源。我们既可以把特征向量直接输入模型（ $\mathbf{x}_i \rightarrow G$ ），也可以先对特征向量中的各个“特征（Feature）”进行相应“预处理（Preprocess）”，然后把预处理后的特征向量作为模型的输入（ $\mathbf{x}_i \xrightarrow{\text{Preprocess}} \mathbf{x}_i^* \rightarrow G$ ）。对特征向量进行预处理这一步通常被称为“数据预处理（Preprocessing）”。此外，一般而言，我们会假设特征向量是 n 维的列向量，即一个特征向量中会有 n 个特征（Feature）：

$$\mathbf{x}_i = (x_i^{(1)}, \dots, x_i^{(n)})^T$$

而对于特征自身而言，则大体上可以分为两种：离散型特征和连续型特征。其中，离散型特征的取值是离散的，它一般只有有限个取值，比如，“是否已经工作”这个特征的取值就只有“是”和“否”这两个取值。而连续型特征的取值就是连

续的，它一般会有无限个取值，比如，我们的身高，当精确到小数点后很多位时，理论上就会有无限个取值

- 标签（Label），它是样本中的 y_i ，是模型的“目标”。我们的最终目的就是能让“特征向量空间”（亦即所有可能的特征向量，常用 \mathbf{x} 指代）中的每个特征向量 \mathbf{x} 在输入我们的模型 G 后，都能输出相对应的、正确的标签 y 。标签同样有离散型标签和连续型标签这两种，比如当 G 是分类器（Classifier）时（输出是某个类别的模型，比如说任务是分辨“猫”和“狗”，那么分类器要么输出“猫”，要么输出“狗”），标签就是离散的，于是“标签空间（即所有可能的标签，常用 \mathbf{y} 指代）”是有限的，这时我们通常会称之为“类别空间”；当 G 是回归器（Regressor）时（输出是某个数值的模型，目的是使模型输出与真实值尽可能接近，比如预测股票价格），标签就是连续的，于是标签空间是无限的。

需要指出的是，在概率论的框架下，我们会将每个特征 $x^{(i)}$ 都视为某个随机变量 $X^{(i)}$ 的采样，从而每个离散型、连续型特征都分别对应着某个离散型、连续型随机变量的采样。此时，我们就能进一步地将特征向量 \mathbf{x} 视为某个随机向量 \mathbf{X} 的采样，其中

$$\mathbf{X} = (X^{(1)}, X^{(2)}, \dots, X^{(n)})^T$$

类似的，我们也会将标签 y 视为某个随机变量 Y 的采样，且当 G 是分类器时， Y 是离散型随机变量；当 G 是回归器时， Y 是连续型随机变量。不难看出，我们想要挖掘的其实正是各个 $X^{(i)}$ 和 Y 之间的关系。

此外，数据集还能进一步划分成如下三大类，这种划分虽然操作起来非常简单，但背后的思想却是重要的。

- 训练集（Training Set），它是我们的模型 G 实际接收到的数据。通常来说，为了使模型能在未知数据上表现得更好（也就是所谓的“泛化能力”更好），我们只会取出数据集中的一部分来作为训练集。不过为了表述方便，我们通常会直接用

$$D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$$

来表示训练集。尽管如此，我们也要记住：在真正完成一项机器学习任务时，训练集一般而言只是数据集的一部分。

- 交叉验证集（Cross Validation Set），它是 G “能看不能用”的数据。具体而言， G 使用训练集来进行训练，并会时不时观察一下它在交叉验证集上的表现来决定是否继续训练，以及是否需要调节它自身的设置。
- 测试集（Test Set），它是用来评估 G 的性能的数据，是 G “看不到”的数据。具体而言，我们只有在使用训练集来训练 G 并使用交叉验证集来监督训练后，才会使用测试集来看看 G 的表现究竟如何。总之，测试集是不会（也绝对不能）参与到训练中的，在实际搭建机器学习模型时一定要注意把它和交叉验证集区分开。

注意：需要指出的是，获取数据集这个过程并不容易。为了在初期快速地验证某个算法所对应的模型 G 的性能，我们不能将时间浪费在收集数据上。在此笔者推荐两个非常著名的、含有大量真实数据集的网站：<http://archive.ics.uci.edu/ml/datasets.html> 和 <https://www.openml.org/>，本书时不时会用到其中的某些数据集来作为样例。

在上文中提到过，我们的最终目的是能让特征向量空间中的每个特征向量在输入我们的模型后，都能输出相对应的、正确的标签。换句话说，我们认为在模型空间中存在着一个“正确的”模型 \hat{G} ，它能将特征向量空间 \mathcal{X} 中的每一个特征向量 \mathbf{x} ，映射到它所应该映射到的、标签空间 \mathcal{Y} 中的某个标签 y 上（我们通常称这个标签 y 为特征向量 \mathbf{x} 的 Ground Truth。翻译成符号语言的话，就是（“ \forall ”表示“任意的”）：

$$\hat{G}(\mathbf{x}) = y \quad (\forall \mathbf{x} \in \mathcal{X})$$

我们的目的，就是让 G 尽可能地去“接近” \hat{G} 。但是机器并不能知道“接近”是什么意思，所以我们需要把它翻译成机器能够听懂的语言。在机器学习领域，我们通常会把“接近”翻译成“将某个损失函数（Loss Function）降到最低”，亦即我们会定义一个损失函数 L ：

$$L(G(\mathbf{x}), \hat{G}(\mathbf{x}))$$

它需要满足“ G 与 \hat{G} 越接近，则对于 $\forall \mathbf{x} \in \mathcal{X}$ ，函数值都越小”这样一个性质，可以看出，损失函数的定义是针对某个局部的。为了“让 G 总是与 \hat{G} 接近”，我们需要定义一个全局的“代价函数（Cost Function）” C ，它是损失函数在 \mathcal{X} 上的期望值：

$$C(G, \hat{G}) = E_{\mathbf{x} \in \mathcal{X}} L(G(\mathbf{x}), \hat{G}(\mathbf{x}))$$

但不难想象的是，我们一般不可能获取到整个样本空间 \mathcal{D} （即一般不可能把所有可能的样本都收集到），所以训练集 D 通常仅仅是 \mathcal{D} 的一个采样。在这种情况下，代价函数就需要重新表述为：

$$C(G, \hat{G}) = \frac{1}{N} \sum_{i=1}^N L(G(\mathbf{x}_i), \hat{G}(\mathbf{x}_i))$$

注意，我们认为 D 中的每个样本都是合理的，即每个特征向量 \mathbf{x}_i 所“应该映射到的标签”都应该是 y_i ，亦即：

$$\hat{G}(\mathbf{x}_i) = y_i$$

所以就有

$$C(G, \hat{G}) = C(G, D) = \frac{1}{N} \sum_{i=1}^N L(G(\mathbf{x}_i), y_i)$$

这种代价函数的重新表述还意味着这样一个事实：我们认为 D 中样本的分布和整个 \mathcal{D} 中样本的分布类似，从而能够通过降低 G 在 D 上的代价来降低 G 在 \mathcal{D} 上的代价。

以回归问题为例，假设我们的目的是学习出二次函数：

$$\hat{G}(x) = (x + 1)^2$$

那么损失函数就可以定义成 G 和 \hat{G} 的输出值的“距离”（平方差）：

$$L(G(x), \hat{G}(x)) = [G(x) - \hat{G}(x)]^2 = [G(x) - (x + 1)^2]^2$$

代价函数就是损失函数在 D 上的“期望”：

$$C(G, \hat{G}) = C(G, D) = \frac{1}{N} \sum_{i=1}^N [G(x_i) - (x_i + 1)^2]^2$$

于是只要模型 G 足够“光滑”，那么该代价函数就是可导的，从而我们就可以通过梯度下降等算法来降低代价函数，进而进行学习。

注意：梯度下降算法的框架会在第2章给出，其衍生的种种优化算法则会在第3章给出，这里暂时按下不表。

接下来说明 SRM 与 ERM 这两个概念，为此我们需要引入“假设空间”和“泛化能力”这两个比较重要而本质的概念。其中，“泛化能力”在前文已有提及，它用于衡量模型 G 在未知数据上的表现。简单来说，如果 G 在训练数据 D 上表现优异，但在 \mathcal{D} 的其余采样（比如交叉验证集或测试集）上表现糟糕，那么我们就说 G 的泛化能力很差，或说此时产生了“过拟合（Over Fitting）”的情况。

一个经典的、可以拿来直观说明过拟合现象的是龙格现象（Runge's Phenomenon），它是用高阶多项式进行多项式插值时出现的问题。具体而言，假设我们的模型 G 为

$$G(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = \sum_{i=0}^n a_ix^i$$

其中 n 为唯一的超参数， $a_0 \sim a_n$ 则是参数。龙格现象是指，当我们试图用上述 G 来拟合

$$f(x) = \frac{1}{1 + 25x^2}$$

这样一个函数时，很容易陷入过拟合的状态，如图 1.1 所示。

在图 1.1 中，每个点都是函数 $f(x)$ 上的一个采样，函数曲线则是当我们把超参数 n 取得比较大时，模型 G 通过多项式插值法拟合出来的结果。可以看到，虽然 G 在 D 的每个样本（即图中的每个点）上都拟合得堪称完美，但从总体上看来毫无疑问是非常糟糕的。

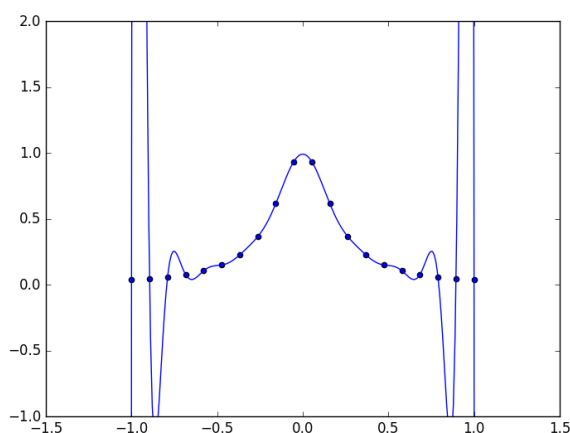


图1.1 过拟合

与过拟合相对的概念叫“欠拟合 (Under Fitting)”，它描述了这样一种情况：即使是在训练集 D 上，模型 G 的表现也非常差，如图 1.2 所示。

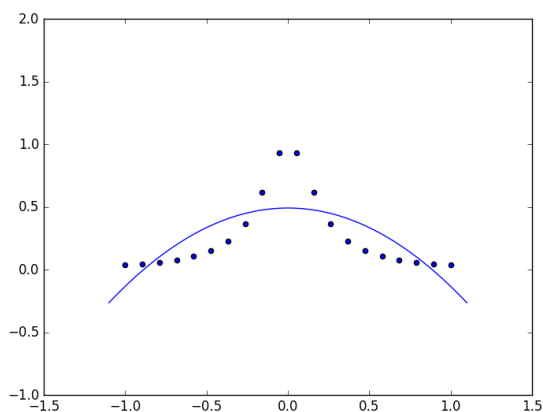


图1.2 欠拟合

注意：并不是说欠拟合时 G 在 D 上的表现就一定好于过拟合时的表现，只是说这两者在训练集 D 上的表现走了两个极端，在 X 其他采样上孰优孰劣是要具体分析。

所以一个很自然的想法就是：我们要“中庸”，要“不偏不倚”，换句话说就是找到过拟合和欠拟合之间的那个平衡点。而这种找平衡点的思想，正是统计学习中“结构风险最小化 (Structural Risk Minimization, 简称 SRM)”的直观解释。与之相对的则是“经验风险最小化 (Empirical Risk Minimization, 简称 ERM)”，它就只注重 G 在 D 上的表现而罔顾 G 在 D 其他采样上的表现，从而意味着它容易陷入过拟合的窘境。用数学语言来说的话，就是 SRM 注重对风险上界的最小化，而不只是像 ERM 那样，单纯地使经验风险最小化。它会从使得风险上界最小的函数子集中，挑选出使得经验风险最小的函数，而不是直接挑选出经验风险最小的函数。

而这个“使得风险上界最小的函数子集”，正是我们之前提到过的假设空间。

SRM 和 ERM 的详细讨论超出了本书的范围，我们这里就只直观地阐述一下相关的理论：

如果能让 G 在只能看到 D 的情况下、能够在训练完之后将性能泛化到整个 \mathcal{D} 上面去的话，我们就不能“放任” G 去自由地训练——即要对 G 加入一定的“限制”，否则随着 G 的能力变得越来越强大，它在 D 上的表现就会越来越完美，此时如果 D 的质量不完美，即不能很好地“代表”整个 \mathcal{D} 的话，就会陷入严重的过拟合。

我们当然不希望出现 G 的能力越强大、最终训练完后反而表现更差的情况，此时上文提到的“限制”就显得尤为重要了。从本质上来说，这个“限制”表现在假设空间的选取上。而从实际操作上来说，一个非常普遍的做法就是对 G 的复杂度做出一定的惩罚，从而使 G 趋于精简。用数学语言来说，就是在定义完算法的代价函数后，加上一项对 G 复杂度的惩罚（我们用 $G_{complexity}$ 来表示 G 的复杂度， $G_{complexity}$ 越大则 G 越复杂）：

$$\tilde{C}(G, \hat{G}) = C(G, \hat{G}) + \lambda G_{complexity}$$

这种惩罚模型复杂度的做法被称为“正则化（Regularization）”，上式中的 λ 常被称为“正则化系数”，它反映了我们对模型复杂度的“惩罚力度”。

相比于通过选取合适的假设空间来规避过拟合，进行交叉验证可以让我们知道过拟合的程度，从而帮助我们选择合适的模型。常见的交叉验证有以下三种。

- **K-fold Cross Validation:** 中文可翻译成 K 折交叉验证，它是应用最多的一种方法。其方法大致如下所示。
 - 将 D 分成 K 份： $D = \{D_1, D_2, \dots, D_K\}$ ，一共做 K 次实验。
 - 在第 i 次实验中，使用 $D - D_i$ 作为训练集， D_i 作为交叉验证集来训练，评测 G 。
 - 最终选择平均交叉验证误差最小的模型。
- **留一交叉验证（Leave-one-out Cross Validation）:** 这是 K 折交叉验证的特殊情况，此时 $K = N$ ，即每次都只从数据集中“留出一个样本”作为评测标准。
- **简易交叉验证:** 这种验证实现起来最简单，也是本书（在进行交叉验证时）最常采用的方法。它简单地将数据进行随机或不随机的分组，最后达到训练集约占原数据的 90% 的程度（这个比率可以视情况改变），选择模型时同样使用评测误差作为标准。

注意：交叉验证的相应实现会在第 7 章给出。

1.2 Python 简介

1.1 节大概介绍了机器学习的各种概念，它们都是相当基础且重要的，后面我们会直接运用相应默认符号并不做额外的说明。这一节我们主要讲讲与脚本语言 Python 相关的一些知识。在 Python 界有一句流传甚广的“谚语”为“人生苦短，我用 Python”，由此也能大概领略到 Python 强大的功能与易于上手的特性。

1.2.1 Python 的优势

正如本章一开始所说，人工智能很有可能是下一次工业革命的驱动力，而 Python 在人工智能领域的竞争力可谓无出其右。从数据的获取（网络爬虫），到数据的清洗、预处理（第三方库 pandas），到模型的搭建（传统机器学习算法库 scikit-learn，以及以 TensorFlow 为代表的各种深度学习框架），到结果的可视化（第三方库 matplotlib），甚至到和服务器的交互（第三方工具 jupyter notebook），Python 都有非常完备的解决方案。

即使是 Python 最坚定的反对者，恐怕也不能否认 Python 编程速度快、能够将开发周期缩短的事实。而机器学习界有一个很有名的观点，就是机器学习任务的解决通常是一个迭代的过程。我们需要在“建模→训练→评估→调参→建模”这个循环中迭代多次后才能获得最终的理想模型。如果使用底层语言（比如 C、C++）的话，“调参”和“建模”的步骤可能会需要比较大的改动才能完成，而由于 Python 非常简单，只要抽象和封装做得足够好，那么上述循环就能以很快的速度迭代起来（本书的代码实现将会体现这一点）。

当然，Python 不可能全无缺点。事实上，它那乌龟一般的执行速度常常被人们抱怨。不过对于人工智能，或者更准确地说，对于深度学习而言，几乎所有成熟的第三方库都使用底层的语言进行核心算法的编写，Python 只是利用了它的“胶水”功能，起到了一个提供接口的作用。除非确实觉得需要从零开始编写一个属于自己的、性能高效的底层算法库，否则如果只是利用算法和对应的模型来写出人工智能的应用的话，没有必要去细究底层究竟发生了什么，而只需把上层的 Python 接口的功能及其背后对应的逻辑弄清楚即可。此外，即使是 Python 本身，如果确实有迫切需要加速的地方，也有许多成熟的解决方案可以采用。比如使用 Cython，它是一种介于 C++ 与 Python 之间的编程语言，能够无缝对接进 Python 程序；再比如我们将会在附录 B 中介绍的 Python 第三方库——numba，它能够通过某种编程范式，使 Python 中的某些操作加速 100 倍以上。

而且，就算我们撇开人工智能不谈，Python 本身也是非常具有魅力的编程语言。开源运动的领袖人物 Eric Raymond 曾经说过：“Python 语言非常干净，设计优雅，具有出色的模块化特性。其最出色的地方在于，它鼓励清晰易读的代码，特别适合以渐进开发的方式构造项目”。对于 Python 程序，人们甚至有时会戏称其为“可执行的伪代码（executable pseudocode）”以突显它的清晰性和可读性。

此外，笔者认为使用 Python 来学习机器学习与编程界金句——“不要过早优化”背后的思路是相通的。通常在检验一个新算法时，我们要做的是进行快速迭代并调优思想而不是优化算法的快慢。事实上，如果思路是错的，那么即使我们把程序的运行效率弄得很高，最终发现一切需要推翻重来时，这些辛辛苦苦做出来的优化很有可能也要跟着推翻重来，这就导致我们做了大量的无用功。所以一般的想法是，先用 Python 做一个原型，在保证思路正确的前提下，如果确实需要更高的效率，再用 C 或 C++ 重写核心部分的代码。更何况对于机器学习而言，诸如 scikit-learn 和 TensorFlow 这种成熟的第三方库已经将效率提到很高了，对于个体而言，一般很难超越它们。

1.2.2 scikit-learn 和 TensorFlow

Python 的强大相当大的部分体现在它那浩如烟海的第三方库上,而 1.2.1 节最后提到的两个第三方库——传统机器学习算法库 `scikit-learn` 和神经网络框架 `TensorFlow` 可谓是最有代表性的。在本节中,我们将简要介绍一下它们的安装与使用。

1. 使用 Anaconda 安装 Python

由于 `scikit-learn` 和 `TensorFlow` 都有不少依赖,一个个地安装这些依赖虽说不是不可以,但却有可能带来不必要的麻烦。这时一个已经集成好大部分科学计算依赖的 Python 环境就显得非常重要,而 `Anaconda` 正是这样一个优质的 Python 集成环境。`Anaconda` 官网的地址为 <https://www.continuum.io/downloads>,该地址对应的页面中央有“Download for”字样,其后跟着的三个图标分别代表 Windows、Mac 和 Linux;单击相应图标,页面就会导航到如图 1.3 所示的部分。

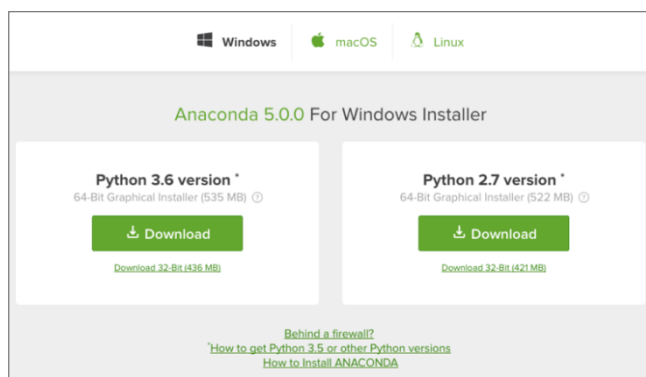


图 1.3 Anaconda 安装包下载页面

本书将使用 Python 3 版本,且笔者强烈建议使用 Python 2 的读者尝试使用 Python 3,因为社区对 Python 2 的支持将在不久之后完全停止。

下载好安装包并完成安装后,就可以进入下一个环节了。

2. scikit-learn 简介与应用

虽然从头开始安装 `scikit-learn` 还是有点折腾的,但幸运的是, `Anaconda` 已经帮我们装好了 `scikit-learn`,所以可以直接使用它。

`scikit-learn` 是一个实现了大部分常用的传统机器学习算法的第三方库,拥有轻便、高效、全面的特点。对于各种耳熟能详的传统机器学习算法,如朴素贝叶斯、决策树、Logistic 回归、SVM 等,在 `scikit-learn` 里面都有相应的实现。

由于后文的相应章节会给出 `scikit-learn` 的具体应用代码,这里作为一个简介,我们就仅仅说明一下应用 `scikit-learn` 的通用流程。

1. 获取数据。该步骤会在 1.3 节中进行较为详细的介绍。在这个步骤之后,我们就拥有训练集 $(x_{\text{train}}, y_{\text{train}})$ 、交叉验证集 $(x_{\text{cv}}, y_{\text{cv}})$ 和测试集 $(x_{\text{test}}, y_{\text{test}})$ 了。

2. 从 `scikit-learn` 中导入相应的模型。以 Logistic 回归这个模型为例，只需一行代码：

```
from sklearn.linear_model import LogisticRegression
```

3. 进行模型的初始化。仍以 Logistic 回归为例，代码如下：

```
clf = LogisticRegression(**kwargs)
```

其中，`kwargs` 是模型的参数。通常来说，`scikit-learn` 设置的默认参数已经足够优异，所以可以直接通过

```
clf = LogisticRegression()
```

进行模型的初始化。

4. 进行模型的训练：

```
clf.fit(x_train, y_train)
```

5. 使用训练好的模型在新数据集上进行预测：

```
y_pred = clf.predict(x_test)
```

把这些代码写在一起的话，就是：

```
01 from sklearn.linear_model import LogisticRegression
02
03 clf = LogisticRegression()
04 clf.fit(x_train, y_train)
05 y_pred = clf.predict(x_test)
```

可以看到整个流程非常简单。事实上，`scikit-learn` 中的大部分模型都可以通过这样几个步骤来运用，可谓方便至极。

注意：Logistic 回归的简介会放在第 2 章第 4 节中。

3. TensorFlow 的简介与安装

一般提及 TensorFlow 时，人们会说它是“深度学习框架”；但正如前文所说，神经网络是深度学习的根基，大部分深度学习模型都离不开神经网络的支撑。虽说 TensorFlow 确实支持实现一些与神经网络无关的模型（本书附录 A 中实现的 LinearSVM 和 SVM 模型就和神经网络毫无关系），但大部分应用还是与神经网络有关的。所以在本书中，TensorFlow 在绝大多数时间内都是作为“神经网络框架”存在的，有关 TensorFlow 的其他应用手段则可以查看相应的资料。

TensorFlow 的安装可以比较简单也可以稍微复杂一点，这取决于你是否打算使用它的 GPU 版本。对于非 GPU 版本（即 CPU 版本）而言，在 Anaconda 环境下安装 TensorFlow 是极其简单的一件事，只需在命令行中输入：

```
pip install --ignore-installed --upgrade tensorflow
```

即可。而对于 GPU 版本来说，首先你需要有一颗支持相应 GPU 运算的 GPU（详情可以参见

<https://developer.nvidia.com/cuda-gpus>), 然后需要下载 CUDA[®] Toolkit 8.0 (<https://developer.nvidia.com/cuda-downloads>) 和 cudnn v6 (<https://developer.nvidia.com/cudnn>), 并把它们加到环境变量中。在安装完这些依赖后, 再在命令行中输入:

```
pip install --ignore-installed --upgrade tensorflow-gpu
```

即可完成 GPU 版本的安装。如果上述说明并不能使你完成安装, 又或是有些步骤的描述过于模糊的话, 可以参见 TensorFlow 官网中的相应说明 (<https://www.tensorflow.org/install/>)。

1.3 前期准备

在运行神经网络乃至所有的机器学习算法时, 我们会有一些“总归要做的事情”, 本节我们会对它们进行介绍, 并在需要的地方提供代码或代码链接。

1.3.1 训练、交叉验证与测试

我们在 1.1.2 节中已经提到了要将数据集划分为训练集、交叉验证集和测试集, 并提供了一些交叉验证的方法, 本节我们会介绍如何用 Python 进行具体的操作。

首先将数据读入 Python。在现实生活中, 数据大致可以分为两种: 结构化数据 (Structured Data) 与非结构化数据 (Unstructured Data), 其中结构化数据可以理解为“行数据”或“存储在表格中的数据”, 比如表 1.1 所示的房价数据就是经典的结构化数据。

表 1.1 房价数据集 (来源: Coursera)

房子面积 (平方英尺)	房子价格 (美元)	房子面积 (平方英尺)	房子价格 (美元)
2104	399900	1600	329900
2400	369000	1416	232000
3000	539900	1985	299900
1534	314900	1427	198999
1380	212000	1494	242500

而所谓的非结构化数据, 就是诸如图片、文档、语音这种数据。由于这些数据的读入方式有很多, 所以为了更好地说明核心问题, 我们接下来会以表 1.1 所示的房价数据集为例, 说明如何将数据集划分为训练集、交叉验证集与测试集。

首先需要知道的是, 在计算机的眼中, 数据集并不是一个表格, 而 (通常) 是一个用某个分隔符 (如逗号 “,”) 分割开的文本文件。以房价数据集为例, 它的前三个样本其实如下所示 (完整的数据集可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Data/prices.txt):

```
2104,399900
1600,329900
2400,369000
```

为了将这种文本文件转换成 Python 中的 numpy 数组（即 numpy 中的 ndarray），我们可以运用如下代码来实现（注：以下代码的运行要求文本文件“Data.txt”位于和程序本体相同的目录中）：

```
01 # 将文本文件 (Data.txt) 转换为 numpy 矩阵
02 import numpy as np
03 with open("Data.txt", "r") as file:
04     data = np.array([line.strip().split(",") for line in file], dtype=np.float32)
```

在把数据集读入 Python 后，我们就可以对它进行划分了。以简易交叉验证为例，如果想要随机抽取 70% 的数据用于训练，我们可以这样做：

```
05 # 计算出训练、交叉验证数据集的大小
06 length = len(data)
07 n_train, n_cv = int(0.7*length), int(0.15*length)
08 # 抽取训练、交叉验证、测试样本的下标
09 idx = np.random.permutation(length)
10 train_idx, cv_idx = idx[:n_train], idx[n_train:n_train+n_cv]
11 test_idx = idx[n_train+n_cv:]
12 # 根据下标直接抽取训练数据集、交叉验证集
13 train, cv, test = data[train_idx], data[cv_idx], data[test_idx]
```

1.3.2 简易数据预处理

注意，上一节将数据读入 Python 时，直接使用了：

```
data = np.array([line.strip().split(",") for line in file], dtype=np.float32)
```

这个在原始数据中每个特征向量 \mathbf{x} 的所有特征均为数值型特征时是可以的，但如果原始数据包含类别型特征，且该特征的取值为字符串时（比如，特征“字母”的取值为“a”“b”……），就不能直接通过指定 `dtype=np.float32` 来完成数据的读取。在这种情况下，我们一般会先将这些字符串转换为数值型数据（比如将“a”转换成“0”、“b”转换成“1”……）。进行数值转换的方法有许多种，这里就介绍一种通用的、效率较高的方法。

```
01 # 定义一个数组来存储 26 个字母
02 # （大家可以想一想：这里这 26 个字母是按照什么规律输入的？）
03 letters = np.array(list("qwertyuiopasdfghjklzxcvbnm"))
04 # 从上述数组中随机抽取 10000 个样本作为我们的数据
05 # 我们的目标是把这个数据转换为数值型数据
06 data = letters[np.random.randint(0, 26, 10000)]
07 # 利用 Python 自带的“集合”数据结构，提取出 data 中不同的特征取值
08 # 在这个问题中，features 即为 {'a' 'b' ... 'z'}
09 features = set(data)
10 # 利用 features 生成“转换字典”
11 # 注意这里我们用了 sorted 以保证 a 对应 0、b 对应 1……
12 feat_dict = {c: i for i, c in enumerate(sorted(features))}
13 # 利用“转换字典”定义将数据转换为数值型数据的“转换函数”
14 def transform(data, feat_dict):
15     return [feat_dict[c] for c in data]
```

做完上面这些工作后，以后即使来了新的数据，只要这个新数据的特征取值范围不超出训练集的特征取值范围（在这个例子中就是 26 个英文字母），我们都可以用转换函数来进行转换。比如：

```
16 new_data = ["a", "c", "b"]
17 print(transform(new_data, feat_dict))
```

将会输出 “[0, 2, 1]”。

在所有数据都被数值化后，我们就能安心地使用 `np.array(data, dtype=np.float32)` 来得到数据矩阵了。不过为了获得更好的结果，我们不能只满足于“对数据进行数值化”这个步骤，而是应该进一步对数据进行处理，这也正是前文提到过的数据预处理。而如果展开叙述的话，数据预处理将是一个庞大的话题，短短的一个小节断然是无法容纳的，所以我们在这里只介绍一种简易有效的数据预处理手段：标准化。具体而言，假设数据集中的连续型特征集为 X ，那么我们将会对它进行如下处理：

$$X \leftarrow \frac{X - \bar{X}}{\text{std}(X)}$$

其中 \bar{X} 表示 X 的均值（Mean）、 $\text{std}(X)$ 表示 X 的标准差（Standard Deviation）。相应的代码实现也很简单：

```
01 x -= x.mean(axis=0)
02 x /= x.std(axis=0)
```

在完成了离散型特征的数值化、连续型特征的标准化后，可以说就完成了简易数据的预处理。需要指出的是，如无特殊说明，在本书后面章节的代码里，我们将统一使用 `x_train`、`y_train`，`x_cv`、`y_cv`，`x_test`、`y_test` 这 6 个（在 1.2.3 节中出现过的）变量来分别代指训练集、交叉验证集和测试集中的特征向量与标签。同时由于我们已经在本节介绍了如何进行数据的读取和预处理，所以为了简便，我们之后一般会直接使用 `x_train`、`y_train` 等变量，而不会给出这些变量的详细定义过程。

注意：更多的数据预处理方法和过程会在第 6 章给出。

1.4 本章小结

- 机器学习打破了传统的工作范式，将核心从硬编程转向了算法的设计、数据的获取与模型的搭建。
- 机器学习中有许多常用的术语，比如“模型”“参数”“超参数”“空间”“数据集”“样本”“特征向量”“标签”……了解它们的具体含义是很重要的。
- 数据集可以分为训练集、交叉验证集与测试集，其中训练集对模型而言“能看且能用”，交叉验证集对模型而言“能看但不能用”，测试集对模型而言则是“不能看只能测”。
- 为了提高模型的泛化能力，通常要对模型的复杂度做出惩罚。

- Python 是一门优秀的语言，代码清晰可读、功能广泛强大。在人工智能领域，Python 有着得天独厚的优势。
- scikit-learn 是一个很好的传统机器学习算法库，TensorFlow 则是一个很好的深度学习框架；在本书中，我们会将 TensorFlow 视为神经网络框架来使用。
- 简易的数据预处理包括两个步骤：
 - 将含有字符串取值的离散型特征数值化。
 - 将连续型特征标准化。

第 2 章

经典传统机器学习算法简介

在正式开始学习神经网络之前，了解一些经典的传统机器学习算法是有必要的——毕竟如果传统机器学习算法已经足够好的话，为什么我们还要大费周折地研究神经网络呢？事实上，正是因为传统的机器学习算法越来越无法满足现代人们的需求，所以以神经网络为根基的深度学习才会如此火爆。

本章打算介绍传统机器学习算法中最为经典的 5 种算法：朴素贝叶斯、决策树、感知机、支持向量机与 Logistic 回归。我们会在本章简要地阐述一下它们的思想和用 `scikit-learn` 实际应用它们的方法，并且在第 3 章介绍完神经网络的基础后，在第 4 章通过将神经网络与它们进行对比来显示出神经网络的优越性。

本章主要涉及的知识点有：

- 朴素贝叶斯算法
- 决策树算法
- 感知机与支持向量机算法
- Logistic 回归

2.1 朴素贝叶斯

朴素贝叶斯算法分为“朴素 (Naïve)”与“贝叶斯 (Bayes)”两个部分，其中“朴素”对应的是条件独立性假设，“贝叶斯”则对应贝叶斯思维。我们会分别在 2.1.1 和 2.1.2 节对它们做出解释，然后在 2.1.3 节介绍朴素贝叶斯算法的具体形式，继而会在 2.1.4 节展示如何具体地应用朴素贝叶斯算法。在最后的 2.1.5 节和 2.1.6 节，我们则会补充说明朴素贝叶斯算法

中的一些细节。

2.1.1 条件独立性假设

在第 1 章我们提到过，通常会假设训练集为

$$D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$$

我们当时没有说明的是，除了一些特殊的问题（比如时序问题）以外，通常还会假设 D 中的各个样本之间总是“独立”的，通俗来说就是样本之间应该是“毫无关系”的。如果用数学语言来描述的话，就是对任意参数 θ 而言，都有：

$$p(D; \theta) = \prod_{i=1}^N p(D_i; \theta)$$

其中

$$D_i = (\mathbf{x}_i, y_i), \quad i = 1, 2, \dots, N$$

注意，我们在第 1 章曾经说过，在概率论的框架下，可以把特征向量 \mathbf{x} 视为随机向量 \mathbf{X} 的采样，其中每个特征 $\mathbf{x}^{(i)}$ 都是随机变量 $X^{(i)}$ 的采样：

$$\mathbf{x} \text{ sampled from } \mathbf{X} = (X^{(1)}, X^{(2)}, \dots, X^{(n)})^T$$

而标签 y 则是随机变量 Y 的采样：

$$y \text{ sampled from } Y$$

所以在概率论的框架下，样本的概率（或概率密度） $p(D_i; \theta)$ 就能很自然地定义出来了：

$$p(D_i; \theta) = p(\mathbf{X} = \mathbf{x}_i, Y = y_i; \theta)$$

为了书写简洁，我们通常会略去随机变量的符号不写，即

$$p(D_i; \theta) = p(\mathbf{x}_i, y_i; \theta)$$

所以“样本之间总是独立的”这个陈述就能表达成

$$p(D; \theta) = \prod_{i=1}^N p(\mathbf{x}_i, y_i; \theta)$$

而条件独立性假设则要更强一些。放到朴素贝叶斯（Naïve Bayes）算法里面来说的话，条件独立性假设阐述的是这样一个假设：对于每个类别来说，特征之间是相互独立的。具体而言，假设每个样本对应的特征向量 \mathbf{x} 有 n 个特征：

$$\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^T$$

那么对于 $\forall y \in \mathcal{Y}$ ，都有

$$\begin{aligned} p(\mathbf{x}|y; \theta) &= p(x^{(1)}, x^{(2)}, \dots, x^{(n)}|y; \theta) \\ &= p(x^{(1)}|y; \theta)p(x^{(2)}|y; \theta) \dots p(x^{(n)}|y; \theta) \\ &= \prod_{i=1}^n p(x^{(i)}|y; \theta) \end{aligned}$$

这个假设和“假设样本之间总是独立的”有一点点像，但本质却完全不同。所以虽然条件独立性假设从直观上来说似乎仍然可以接受，但已经是一个事实上很难满足的假设（或说是一个很强的假设）了，这也是“朴素”二字的来由。

2.1.2 贝叶斯思维

贝叶斯思维是很模糊且不严谨的说法，它通常代指这样一种思想：我们需要在一个问题的解决方案中加入“先验知识（Priori）”，然后根据该“先验知识”进行“推理（Inference）”并总结出“后验知识（Posteriori）”。

这段话涉及贝叶斯思维中很重要的三个概念，我们可以这样直观地去理解它们：

- 所谓的“先验知识”，就是将其视为“真理”的东西。
- 所谓的“推理”，就是在承认先验知识为真理的前提下，做出的合理推断。
- 所谓的“后验知识”，就是推理后得到的、用于进行最终决策的东西。

可能读者会问：这种思维难道不是显而易见的吗？为什么还要弄一个玄乎其玄的名字——“贝叶斯思维”呢？事实上，在对概率的解释这个层面上，我们平时的思维可能还真不是这样的。举一个简单的例子：如果我告诉你，一支足球队在五场比赛中赢了三场，让你估计这支球队在未来十场比赛中的胜率，你会怎么估计？可能许多人的第一反应就是：那我就估计它未来的胜率是 60% 吧！这当然是一个经典、合理的估计。不过可能有些人转念一想，就会问我，这支球队到底是哪一支球队，然后根据他们对该球队的了解，来微调“60%”这个胜率（事实上在笔者写下这段话时，刚好就有一个类似的真实例子：在 NBA 中，上个赛季刚拿了总冠军、且在休赛期补强了的金州勇士队，在新赛季的头三盘中输了两盘）。

在上面这个例子中，“对球队的了解”就是先验知识，“通过了解来微调决策”的过程就是推理，最终得出的胜率就是后验知识。直观地说，这样得到的胜率会比单纯地估计 60% 要靠谱不少。

但是，并不是说贝叶斯思维就是十全十美的。仍然以预测胜率为例，假如我们去问一个伪球迷，他对球队其实只是一知半解，甚至对球队有着深深的误解，那么他在错误的先验知识下推理出来的后验知识，很有可能会比“简单的”60%要差。换句话说，贝叶斯思维的有效性极度依赖于先验知识的有效性。

注意：我对这里“简单的”三个字加了双引号，是因为它虽然非常符合我们的直观理解，但它背后的理论（极大似然估计）却相当基础、重要且不平凡。极大似然估计我们会在后文进行说明，这里暂时按下不表。

总结一下，不难得出这样的结论：

- 当拥有有效的先验知识时，使用贝叶斯思维通常会使决策更合理。
- 当先验知识不足时，使用“简单的”决策一般反而会更合理。

不过对于一个现实中的复杂机器学习任务而言，我们很难或者说根本不知道我们的先验知识到底合不合理，这时究竟要不要引入先验知识就成了大问题。事实上，统计学家之所以分成“频率学派”和“贝叶斯学派”，原因之一就是在这个大问题上分歧——因为即使抛开现实意义不谈，单从理论的角度来看，这个问题也牵扯到了“概率的解释”这个最基本的问题。贝叶斯学派的想法我们上面已经说了不少，他们的核心思想在于引入先验知识并推理出后验知识，即概率（后验知识）随着先验知识和推理手段的改变而改变。而对于频率学派而言，他们认为在解释概率时不应该用到先验知识，概率应该是“自然存在着的”东西。仍以预测胜率为例，频率学派会认为，该球队的胜率是一个固定的数，我们能做的只是拿现有的数据去估计出这个“真实的”胜率。频率学派的核心思想在于，当数据量足够大时，只要我们的估计是无偏的，那么大数定律就可以保证我们获得这个真实的胜率。

总结一下，贝叶斯学派认为当先验知识足够完备时，一件事情的发展趋势就是确定的：我们通过恰当的推理后，就能够得到完备的后验知识。而当先验知识不足时，一件事情的发展趋势充满了随机性：此时即使再怎么推理，也只能得到带有随机性的后验知识。而频率学派则认为，一件事情的发展趋势的随机性天生固有，我们只能通过不断地观测来估计出该随机性的性质，而且只要观测得足够多，这种估计就能达到任意高的精度。

让我们来看一个小例子以直观理解这两者的区别。假设现在小明扔了一个普通的色子，在离他 30cm、50cm 和 100cm 处分别有三个人，然后小明在色子显示出结果后迅速把色子盖上并询问这三个人他扔出了几点。不难想象，不管小明再怎么迅速地盖上，这三个人都或多或少地看到了一些信息，这些信息就是“先验知识”。对于频率学派而言，这些信息的存在与否是无关紧要的，重要的是概率本身，所以会认为这三个人猜对的概率都是 $1/6$ 。但对于贝叶斯学派而言，后验知识（即猜小明扔出了几点）应该由先验知识经过推理后得出。所以假设离小明最近的人看到了“骰子向上的那一面有一个红色的大点”这样一个先验知识，那他自然应该认为小明扔出了 1，且该后验知识是绝对正确的、不存在随机性的。而假设离小明最远的人只看到了“色子向上的那一面的点排成了一排”这样一个先验知识，那对他而言就会从 1、2、3 中选取，此时他猜对的概率就是 $1/3$ 。

2.1.3 模型算法

前文讲了很多思想上的东西，本节我们就要落实到实际理论上了。首先我们要阐述概率模型的概念，它和普通模型相比，输出的会是一个概率值而不是一个用于分类或回归的、不具备

概率意义的数值。换句话说，假设我们现在把输入数据分成三类，普通的模型在接受一个特征向量后，输出的可能就是 1、2 或 3，即输出特征向量所应该属于的类别，概率模型输出的却是该特征向量属于第 1、2、3 类的概率。

用符号语言来说的话，假设现在共有 K 个类别 (c_1, c_2, \dots, c_K) ，输入的特征向量为 \mathbf{x} ，概率模型为 P ，相应的普通模型为 G ，模型的参数为 θ ，则

$$P(\mathbf{x}; \theta) = \begin{bmatrix} p(y = c_1 | \mathbf{x}; \theta) \\ p(y = c_2 | \mathbf{x}; \theta) \\ \vdots \\ p(y = c_K | \mathbf{x}; \theta) \end{bmatrix}$$

且一般会有

$$G(\mathbf{x}; \theta) = \arg \max_k P(\mathbf{x}; \theta)$$

注意：在二分类问题下，通常不会直接使用 $G(\mathbf{x}; \theta) = \arg \max_k p(\mathbf{x}; \theta)$ 这个公式来决定 G 的输出。我们会在附录 E 中进行相应的讨论，这里大家只需留个印象即可。

由上述定义不难看出，如果给定了某个特定的标签 $y \in \mathcal{Y} = \{c_1, c_2, \dots, c_K\}$ 的话，概率模型 P 就可以写成

$$P(\mathbf{x}, y; \theta) = p(y | \mathbf{x}; \theta)$$

于是 G 也可以相应地写成

$$G(\mathbf{x}; \theta) = \arg \max_y P(\mathbf{x}, y; \theta) = \arg \max_y p(y | \mathbf{x}; \theta)$$

之所以要把模型写成这种形式，主要是为了引入贝叶斯公式。我们在 2.1.1 节中就已经提到了朴素贝叶斯中“朴素”二字的由来，但对于“贝叶斯”的由来却还没有说。事实上，“贝叶斯”这三个字在思想上代指的自然是贝叶斯思维，在形式上代指的则是贝叶斯公式：

$$p(\mathbf{x})p(y | \mathbf{x}; \theta) = p(\mathbf{x}, y | \theta) = p(y)p(\mathbf{x} | y; \theta) \Rightarrow p(y | \mathbf{x}; \theta) = \frac{p(y)p(\mathbf{x} | y; \theta)}{p(\mathbf{x})}$$

利用贝叶斯公式， G 就可以进一步改写成：

$$G(\mathbf{x}; \theta) = \arg \max_y \frac{p(y)p(\mathbf{x} | y; \theta)}{p(\mathbf{x})} = \arg \max_y p(y)p(\mathbf{x} | y; \theta)$$

至此，其实我们已经完成了朴素贝叶斯的核心推导之一了，上面这个公式中的 G ，其实就是参数为 θ 下的朴素贝叶斯模型。不难看出，剩下的难点就在于如何估计 $p(y)$ 和 $p(\mathbf{x} | y; \theta)$ 这两个参数的值，而估计的过程其实正是朴素贝叶斯的另一个核心。由于相应的理论虽然很直观但是相应的理论却比较复杂，我们就先在这里提供一个最终的（相当直观的）结果，相关的细节在 2.1.5 节将作为附加内容进行探讨。

我们说起估计概率时，可能马上想到的就是“用频率去估计概率”。比如，一支球队在五场比赛中赢了三场，我们就倾向于认为它未来的胜率是 $3/5$ ，也就是 60% 。这种频率估计概率的思想同样适用于朴素贝叶斯算法。具体而言，假设现在有一个含 N 个样本的数据集 $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$ ，其中有 N_1 个样本属于第 1 类（即标签为 c_1 ）、 N_2 个样本属于第 2 类（即标签为 c_2 ），且 $N_1 + N_2 = N$ （即总共只有两类），那么我们就可以直接估计

$$p(y = c_1) = \frac{N_1}{N}, \quad p(y = c_2) = \frac{N_2}{N}$$

为了方便叙述，我们把 $y = c_1$ 这个事件简写为 y_1 、 $y = c_2$ 这个事件简写为 y_2 ，那么就有

$$p(y_1) = \frac{N_1}{N}, \quad p(y_2) = \frac{N_2}{N}$$

当问题拓展到有 K 个类别时，可以直接估计

$$p(y_k) = \frac{N_k}{N}, \quad k = 1, 2, \dots, K$$

而朴素贝叶斯中的另一项—— $p(\mathbf{x}|y; \theta)$ 估计起来就稍微麻烦一点，但丝毫不改其“用频率估计概率”的本质。同时由于朴素贝叶斯还有一个条件独立性假设，所以该项还可以继续进行分解（假设特征向量 \mathbf{x} 有 n 个特征）：

$$p(\mathbf{x}|y; \theta) = \prod_{i=1}^n p(x^{(i)}|y; \theta)$$

于是我们要做的就是估计出每个分项 $p(x^{(i)}|y; \theta)$ 的值。在频率估计概率的指导下，不难得出这样的算法：假设现在第 k 类（即 $y = c_k$ ）中， $x^{(i)}$ 这个特征出现了 $N_k^{(i)}$ 次，由于第 k 类总共有 N_k 个样本，所以就可以直接估计：

$$p(x^{(i)}|y_k; \theta) = \frac{N_k^{(i)}}{N_k}$$

该估计可以适用于 $\forall k = 1, 2, \dots, K$ 和 $\forall i = 1, 2, \dots, n$ ，于是所有的 $p(x^{(i)}|y; \theta)$ 都能估计出来，于是 $p(\mathbf{x}|y; \theta)$ 也就能估计出来了。

但是，不难看出上面这种叙述有一个问题：我们究竟应该如何从数学上去定义“ $x^{(i)}$ 这个特征出现了 $N_k^{(i)}$ 次”这件事？解决方案比较简单粗暴：先假设特征 $x^{(i)}$ 都是二值特征（即只有 0 和 1 这两个取值），然后认为：

- $x^{(i)} = 1$ 意味着特征 $x^{(i)}$ 出现了。
- $x^{(i)} = 0$ 意味着特征 $x^{(i)}$ 没出现。

此时，条件概率 $p(x^{(i)}|y_k; \theta)$ 的直观意义就变成了“特征 $x^{(i)}$ 在 y_k 的条件下出现的概率”；那么再结合特征 $x^{(i)}$ 本身两个取值的直观意义，我们就可以认为：

- $x^{(i)} = 1$ 时, 条件概率 $p(x^{(i)}|y_k; \theta)$ 需要对整体的条件概率 $p(\mathbf{x}|y; \theta)$ 做出贡献。
- $x^{(i)} = 0$ 时, 条件概率 $p(x^{(i)}|y_k; \theta)$ 不应影响整体的条件概率 $p(\mathbf{x}|y; \theta)$ 造成影响。

所以, 总的条件概率公式应该写成:

$$p(\mathbf{x}|y; \theta) = \prod_{i=1}^n p(x^{(i)}|y; \theta)^{x^{(i)}}$$

也就是说, 应该在每个特征的条件概率上加一个特征本身的次方, 从而当特征 $x^{(i)}$ 没出现($x^{(i)} = 0$)时, 相应的条件概率对总体条件概率不做出贡献($p(x^{(i)}|y; \theta)^{x^{(i)}} = 1$), 当特征出现时才对总体条件概率做出相应的贡献。

综上所述, 不难写出朴素贝叶斯的算法(参见算法 2.1), 该算法对应的模型通常被称为 MultinomialNB。

算法 2.1 朴素贝叶斯算法

输入: 训练数据集 $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$

过程:

(1) 估计先验概率 $p(y_k)$:

$$p(y_k) = \frac{N_k}{N}, k = 1, 2, \dots, K$$

(2) 估计条件概率 $p(x^{(i)}|y_k; \theta)$:

$$p(x^{(i)}|y_k; \theta) = \frac{N_k^{(i)}}{N_k}$$

输出: 朴素贝叶斯模型, 能够估计数据 \mathbf{x}^* 的类别:

$$y^* = G(\mathbf{x}^*) = \arg \max_k p(y_k) \prod_{i=1}^n p(x^{*(i)}|y_k; \theta)^{x^{*(i)}} = \arg \max_k \frac{N_k}{N} \prod_{i=1}^n \left(\frac{N_k^{(i)}}{N_k} \right)^{x^{*(i)}}$$

这个算法当然还是有诸多问题的。除了刚刚做的一个看上去很强的假设(假设每个特征 $\mathbf{x}^{(i)}$ 都是二值特征)以外, 我们还无法处理连续型特征的问题, 因为对于连续型特征而言, 我们基本不能用频率来估计概率。不过有趣的是, 如果所有特征都是离散型特征的话, 那么通过某种转换, 就能把原特征对应的朴素贝叶斯模型等价于一个只有二值特征的朴素贝叶斯模型。不过由于这个转换过程本身以及连续型特征的处理都不直接与本书的主题相关, 所以我们会在 2.1.6 节将它们作为附加内容进行相应的说明, 感兴趣的读者可以阅读相应的部分。而对于本书主题而言, 其实我们只需记住这样两个结论即可:

- 对于特征全是离散型特征的数据集, 可以通过一些手段来使得数据适配于算法 2.1, 从而搭建出相应的朴素贝叶斯模型。
- 对于有连续型特征的数据集, 也可以通过一些手段来将相应特征离散化, 从而也能运用算法 2.1 来搭建出相应的朴素贝叶斯模型。

2.1.4 实例演示

本节我们会展示 MultinomialNB 的应用方法, 将使用笔者比较喜欢拿来做实例的、UCI 上

比较出名的“蘑菇数据集 (Mushroom Data Set)”来进行相应的实验与评估。之所以总是先选用该数据集是因为它具有如下几个特点：

- 该数据集是二分类问题（判断一个蘑菇是否有毒），比较具有代表性。
- 该数据集的大小和特征个数都适中（8124 个样本，22 个属性），且数据很干净，几乎不用进行额外的数据预处理工作。
- 虽然我们人类直接对该数据集进行分类是比较困难的，但是对于机器学习模型来说，该数据集是非常简单的，由此可以直观地感受到机器学习的强大。

而由于该数据集的所有特征都是离散型的，所以它更是非常适合用来测试算法 2.1 所对应的 MultinomialNB 模型。完整的数据集可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Data/mushroom.txt（第一列数据是类别）。下面我们展示一下整个问题解决流程的代码（由于我们已经在 1.3.2 节中展示了如何将数据数值化，所以略去相应步骤并认为已经得到了数值化后的数据：x_train、y_train、x_test 和 y_test）。

为了应用 scikit-learn 中的 MultinomialNB，我们需要先对原始数据做 OneHot Encoding（OneHot Encoding 的定义和必要性的讨论会放在 2.1.6 节）：

```
01 # 从 scikit-learn 中调用相应的预处理器
02 from sklearn.preprocessing import OneHotEncoder
03 enc = OneHotEncoder()
04 # 调用 fit_transform 来对输入数据做 OneHot Encoding
05 x_train_one_hot = enc.fit_transform(x_train)
06 # 由于上一步已经把所需的数据信息收集完毕
07 # 所以这一步只需直接 transform 即可完成转化
08 x_test_one_hot = enc.transform(x_test)
```

当然，其实我们也可以自己实现一个 OneHot Encoder（对数据做 OneHot Encoding 的工具），难度也不是很高：

```
01 # feature 代指我们想要做 OneHot Encoding 的某个特征
02 # n 代指该 feature 可能的取值个数
03 def get_one_hot(feature, n):
04     # 将 OneHot 矩阵初始化为 0 矩阵
05     one_hot = np.zeros([len(feature), n])
06     # 将相应位置上的元素置为 1，完成对 feature 的 OneHot Encoding
07     one_hot[range(len(one_hot)), np.asarray(feature, np.int)] = 1
08     return one_hot
```

但 scikit-learn 中的 OneHot Encoder 转化出来的 OneHot 矩阵有一个很好的性质：它的类型是 scipy 中的 sparse 矩阵，也就是说，它能享受 scikit-learn 中各种算法（尤其是像朴素贝叶斯这样的经常处理稀疏数据的算法）对稀疏输入的优化（这在自然语言处理方面显得尤为重要，比如用朴素贝叶斯做垃圾邮件分类）。

言归正传，现在有了经过 OneHot Encoding 的输入，接下来要做的就很简单了：

```
09 # 从 scikit-learn 中调用 MultinomialNB
10 from sklearn.naive_bayes import MultinomialNB
```

```

11
12 clf = MultinomialNB()
13 # 调用 fit 函数进行训练
14 clf.fit(x_train_one_hot, y_train)
15 # 调用 predict 函数进行预测，然后通过一些 numpy 运算输出模型的准确率
16 print(np.mean(y_test == clf.predict(x_test_one_hot)))

```

得到的结果是 0.953389830508，即准确率在 95.34% 左右，算是一个不错的结果。

如果对 `sklearn.naive_bayes` 做一些调查的话，你会发现它还包含一个叫 `GaussianNB` 的模型，这个模型是干什么的呢？事实上不难发现，目前为止我们所讨论的问题的范围其实还很窄：我们假设所有特征都是离散型的。`GaussianNB` 的目标就是解决另一个“极端”的问题——它会假设所有特征都是连续型特征。

`GaussianNB` 的应用方法和 `MultinomialNB` 的应用方法几乎一致，只需将第一行的

```
01 from sklearn.naive_bayes import MultinomialNB
```

换成

```
01 from sklearn.naive_bayes import GaussianNB
```

即可。

看到这里，读者想必会问：那如果我既有离散型特征，又有连续型特征的话，应该怎么应用朴素贝叶斯算法呢？我们会在 2.1.6 节给出相应的讨论。但是由于相应的实现比较复杂，且它不是我们关注的重点，这里就不展开叙述了，感兴趣的读者可以参见笔者的纯 `numpy` 实现 (https://github.com/carefree0910/MachineLearning/blob/Book/b_NaiveBayes/Vectorized/MergedNB.py)。

如果是单纯的 `MultinomialNB` 的实现的话，还是比较简单的，总代码量只有 30 行左右，有需要的读者可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/b_TraditionalML/MultinomialNB.py。由于笔者的这个实现用到了一些 `numpy` 的特殊技巧，所以在这里就不进行展示了。

2.1.5* 参数估计

本节将对 2.1.3 节中没有展开叙述的、和本书主题相关性没那么紧密的细节做一些补充说明。可以看到节序号右上角加了一个星号，如前言所说，这代表着当前节是附加章节，它有助于读者更深刻地理解一些知识背后的理论，但跳过它们不看也不会对理解本书主题造成很大的影响。

下面就来看看本书的第一个附加内容：如何进行参数的估计。我们在 2.1.3 节中直观地说明了估计的方案——使用频率来估计概率，在本节中将会给出证明，这种估计其实就是频率学派中的极大似然估计（Maximum Likelihood Estimation，常简称为 MLE）。

所谓的 MLE，基于这样一个思想：对于给定的概率模型 P 和训练集 D ，我们可以算出“ D 在 P 上的概率”：

$$P(D|\theta) = p(D_1, D_2, \dots, D_N|\theta) = \prod_{i=1}^N p(D_i|\theta)$$

其中我们用到了前文提到过的“认为样本之间总是相互独立”的这样一个假设。MLE 认为，最好的参数 $\hat{\theta}$ 应该能使得（质量足够好的、具有代表性的）训练集 D 在样本空间 \mathcal{D} 中出现的概率达到最大值，即

$$\hat{\theta} = \arg \max_{\theta} P(D|\theta) = \arg \max_{\theta} \prod_{i=1}^N p(D_i|\theta)$$

由于直接对连乘式

$$\prod_{i=1}^N p(D_i|\theta)$$

求最大值（虽然可行但）比较复杂，我们通常会将问题转化为求对数最大值，即

$$\hat{\theta} = \arg \max_{\theta} \log \prod_{i=1}^N p(D_i|\theta) = \arg \max_{\theta} \sum_{i=1}^N \log p(D_i|\theta)$$

这就是极大似然估计的一般形式，我们常称式中的 $p(\cdot|\theta)$ 为似然函数，它能计算出某个样本在样本空间中出现的概率。

注意：这种运用对数函数把连乘式转为连加式，从而把问题简化的思想非常重要，我们在第 4 章还会用到这种技巧。

为了更好地理解 MLE，我们再来看一下扔色子这个问题。假设现在不知道色子有多少个面，只知道色子在扔出去之后各面出现的概率均相等。此时，色子的面数即为待估计的参数（不妨设为 k ），每一次扔出的点数即为样本 D_i （注意此时的样本里没有特征向量，只有标签，亦即 $D_i = y_i$ ），且似然函数 p 需要满足这样一个性质：

$$p(y_i = 1|k) = p(y_i = 2|k) = \dots = p(y_i = k|k), \quad \forall i$$

即色子 k 个面上的点数在每次扔完色子之后出现的概率都是一样的。假设我们想通过扔很多次色子（比如 $6N$ 次）并记录点数 1 出现的次数来估计色子的面数，然后发现在 $6N$ 次扔色子的过程中，点数 1 总共出现了 N 次。从频率估计概率的角度来看，似乎估计色子有 6 面是一个不错的选择，下面我们就来说明该估计就是极大似然估计。

按照极大似然估计的一般形式，我们需要求

$$\hat{k} = \arg \max_k \sum_{i=1}^{6N} \log p(D_i|k) = \arg \max_k \left[\sum_{i=1}^N \log p(y_i = 1|k) + \sum_{i=1}^{5N} \log p(y_i \neq 1|k) \right]$$

注意：这里我们省去了一些常数项，因为它们不影响 argmax 的结果。

由于

$$p(y_i = 1|k) = p(y_i = 2|k) = \cdots = p(y_i = k|k), \quad \forall i$$

且由概率本身的性质可知

$$p(y_i = 1|k) + p(y_i \neq 1|k) = \sum_{j=1}^k p(y_i = j|k) = 1, \quad \forall i$$

从而

$$p(y_i = 1|k) = \frac{1}{k}, \quad p(y_i \neq 1|k) = \frac{k-1}{k}, \quad \forall i$$

于是

$$\begin{aligned} \hat{k} &= \arg \max_k \left[\sum_{i=1}^N \log \frac{1}{k} + \sum_{i=1}^{5N} \log \frac{k-1}{k} \right] \\ &= \arg \max_k [-N \log k + 5N \log(k-1) - 5N \log k] \\ &= \arg \max_k [-6N \log k + 5N \log(k-1)] \end{aligned}$$

对上式求导，即可得

$$-\frac{6N}{\hat{k}} + \frac{5N}{\hat{k}-1} = 0 \Rightarrow \hat{k} = 6$$

从而完成了证明。这个证明虽然不难，但已经包含了 MLE 的几大关键步骤：

- 求出总的对数似然函数。
- 若有约束（比如概率和要为 1），则根据约束条件来获得更多的信息。
- 综合上述两点，通过求导、解方程来最终得到参数的极大似然估计。

在扔色子这个问题中，约束条件非常简明，所以我们可以直接得到很强的信息。对于朴素贝叶斯而言，它的约束条件就会稍微难利用一点（需要用到拉格朗日方法）。不过利用同一套方法，确实可以证明算法 2.1 中的参数估计就是 MLE，感兴趣的读者可以尝试自己进行推导，也可以参见这篇文章：<http://www.carefree0910.com/posts/e312d61a/>。

除了 MLE 以外，我们还有许多其他的参数估计方法，比如著名的极大后验概率估计（Maximum a Posteriori Estimation，常简称为 MAP 估计）。之所以要提到这个 MAP 估计，是因为虽然我们在 2.1.3 节中说朴素贝叶斯蕴含了贝叶斯思维，但从上下文来看却好像没有体现出这一点。所以下面我们就来说明，朴素贝叶斯在进行决策时用的是 MAP 估计，而 MAP 估计正是贝叶斯思维的很好的体现。

首先来阐述 MAP 估计的一般形式。相比较 MLE 的公式

$$\hat{\theta} = \arg \max_{\theta} P(D|\theta)$$

而言，MAP 估计认为我们应该对参数 θ 有一个先验知识（贝叶斯思维的体现），所以我们应该最大化的是基于该先验知识（利用 D ）经过推理后得到的后验知识：

$$\hat{\theta}_{MAP} = \arg \max_{\theta} P(\theta|D)$$

利用贝叶斯公式，即得

$$\hat{\theta}_{MAP} = \arg \max_{\theta} \frac{p(\theta)p(D|\theta)}{p(D)} = \arg \max_{\theta} p(\theta)p(D|\theta)$$

应用对数简化的思想，就有

$$\hat{\theta}_{MAP} = \arg \max_{\theta} [\log p(\theta) + \log p(D|\theta)]$$

这就是 MAP 估计的一般形式。回忆上文，我们说过贝叶斯模型的决策过程可以表达为

$$G(\mathbf{x}; \theta) = \arg \max_y p(y)p(\mathbf{x}|y; \theta)$$

写成对数的形式，就是

$$G(\mathbf{x}; \theta) = \arg \max_y [\log p(y) + \log p(\mathbf{x}|y; \theta)]$$

可以看到这和 MAP 估计的形式一模一样。从理论层面上来看的话，只需要把标签 y 视为参数的一部分即可。这其实是相当直观的：在做决策时，我们通常会看每一类标签 y 在概率模型 P 下输出的概率，然后根据这些概率来决定最终预测为哪个类。而在看某个 y 在 P 下输出的概率时， y 就已经是给定了的，所以它确实应该作为参数的一部分存在。

2.1.6* 朴素贝叶斯的改进

2.1.3 节中叙述的朴素贝叶斯（算法 2.1）有两个假设：一是假设了所有特征都是二值特征；二是假设了所有特征都是离散型特征。这两个假设在真实数据集上基本是不可能被满足的，所以我们需要做一些数据预处理或直接对算法本身做出改进。

先来看第一个问题该如何解决（即仍然假设所有特征都是离散型特征）。假设现在有一个特征 $\mathbf{x}^{(i)}$ ，它总共有 $n^{(i)}$ 个可能的特征取值，即

$$\mathbf{x}^{(i)} \in \{\mathbf{x}^{(i)(1)}, \mathbf{x}^{(i)(2)}, \dots, \mathbf{x}^{(i)(n^{(i)})}\}$$

之前之所以做了“所有特征都是二值特征”的假设，从直观上来说的话，是因为当时我们认为“ $\mathbf{x}^{(i)}$ 出现了 $N_k^{(i)}$ 次”，这样的事件只有在 $\mathbf{x}^{(i)}$ 是二值特征时才比较方便用数学的语言去表达，当 $\mathbf{x}^{(i)}$ 是多值特征时似乎很难去良定义它。下面我们就展示一种转换手段来解决该问题，它会利

用一些构造出来的二值特征来辅助我们做出相应的定义。

具体而言，其实只需要把该特征展开成一个 $n^{(i)}$ 维的 OneHot 向量 $\tilde{\mathbf{x}}^{(i)}$ 即可。所谓的 OneHot 向量，是指除了某一维的取值是 1 以外，其他维度都取值为 0 的向量。比如，三维的 OneHot 向量就只有如下三种：

$$[0,0,1], \quad [0,1,0], \quad [1,0,0]$$

在知道 OneHot 的定义后， $\mathbf{x}^{(i)} \rightarrow \tilde{\mathbf{x}}^{(i)}$ 的转化过程就能很自然地定义出来了：

$$\mathbf{x}^{(i)} = x^{(i)(j)} \Rightarrow \tilde{\mathbf{x}}^{(i)} = [0, \dots, \overset{j}{1}, \dots, 0]$$

即当 $\mathbf{x}^{(i)}$ 取到第 j 个值时，对应的 OneHot 向量 $\tilde{\mathbf{x}}^{(i)}$ 就在第 j 维取 1。我们通常把离散型特征转化为 OneHot 向量的过程称作“OneHot Encoding”，它是处理离散型特征的一个重要手段，在许多场景中都会应用到它（比如将会在 2.2 节介绍的决策树中）。

在完成 OneHot Encoding 后，从直观上来说，我们就能很自然地定义出“多值特征 $\mathbf{x}^{(i)}$ 出现了 $N_k^{(i)}$ 次”这样的事件了——只需把特征 $\mathbf{x}^{(i)}$ 展开成 $\tilde{\mathbf{x}}^{(i)}$ 即可。此时，由于 $\tilde{\mathbf{x}}^{(i)}$ 是一个 OneHot 向量，所以可以视其为 $n^{(i)}$ 个二值特征的组合，从而此时就相当于将事件“多值特征 $\mathbf{x}^{(i)}$ 出现了 $N_k^{(i)}$ 次”也展开成了 $n^{(i)}$ 个“二值特征 $\mathbf{x}^{(i)(j)}$ 出现了 $N_k^{(i)}$ 次， $j = 1, 2, \dots, n^{(i)}$ ”这样可以被良定义的事件，这就意味着事件“多值特征 $\mathbf{x}^{(i)}$ 出现了 $N_k^{(i)}$ 次”本身也是可以良定义的。

对上文这些讨论做一个总结的话，我们就能解释 2.1.4 节遗留的那个问题了：为什么在 scikit-learn 里面使用 MultinomialNB 时要求先对输入做 OneHot Encoding？因为 scikit-learn 实现的 MultinomialNB 算法正是我们的算法 2.1，所以需要输入做 OneHot Encoding 来将问题转化为算法 2.1。

然后来看第二个问题该如何解决。为了简化问题并抓住本质，不妨先假设此时所有特征都是连续型特征，然后再看特征种类混合的情况。

我们在前文曾经说过，朴素贝叶斯算法的关键在于估计出类别的先验概率 $p(y)$ 和条件概率 $p(\mathbf{x}|y; \theta)$ 。而由于 $p(y)$ 的估计和特征向量 \mathbf{x} 没有关系，所以我们关心的是当 \mathbf{x} 的各个特征都是连续型特征时，应该如何估计 $p(\mathbf{x}|y; \theta)$ 。

在算法 2.1 的指导下，不难得到这样一个直观的估计方法：使用小区间来对 \mathbf{x} 的每个特征 $\mathbf{x}^{(i)}$ 的取值区间进行切割，从而直接使其离散化。具体而言，假设 $\mathbf{x}^{(i)}$ 是一个分布在-1到 1 这个区间内的连续型特征，那么就可以定义这样一个函数：

$$f(\mathbf{x}) = k, \quad \text{if } \frac{k-1}{10} < \mathbf{x} \leq \frac{k}{10}$$

令 $\tilde{\mathbf{x}}^{(i)} = f(\mathbf{x}^{(i)})$ ，不难看出 $\tilde{\mathbf{x}}^{(i)}$ 就是 $\mathbf{x}^{(i)}$ 离散化后的特征。在对所有这种连续型特征都离散化后，我们就能用算法 2.1 来进行模型的搭建了。

这种离散化的方法虽说也有一定的作用，不过如果想获得好的结果，一般需要比较精细地

控制小区间的大小，而这通常是相当困难的。此外，离散化方法对训练集质量的要求也比较高，否则一个在训练集上表现得不错的小区间切割，放到测试集上可能就会一塌糊涂。所以在实际情况中，除非对数据集有很深刻的认识、知道用小区间切割会获得特殊性质以外，我们一般会选用第二种方法：假设该连续型特征服从某个分布，然后把估计概率这个问题转化为估计那个分布的参数（参数估计的手段仍然是 MLE）。而由于现实世界中的每个事件都拥有很多的影响因素，由中心极限定理可知，假设该连续型特征服从高斯分布（Gaussian Distribution，它有一个更通俗的名字——正态分布）是一个比较合理的选择。事实上，2.1.4 节中所说的 scikit-learn 中的 GaussianNB 模型之所以叫这个名字，正是因为它假设了所有连续型特征都服从高斯分布。

而所谓的高斯分布，其实是这样的东西：若某个随机变量 X 服从于均值为 μ 、方差为 σ^2 的高斯分布，即若

$$X \sim N(\mu, \sigma^2)$$

那么就有

$$p(X = x | \mu, \sigma^2)$$

所以一个很自然的想法就是，如果我们的连续型特征 $x^{(i)}$ 所对应的随机变量 $X^{(i)}$ 在标签 $y = c_k$ 的条件下，服从均值为 μ_{ik} 、方差为 σ_{ik}^2 的正态分布，那么就可以直接认为：

$$p(x^{(i)} | y = c_k; \theta) = p(x^{(i)} | \mu_{ik}, \sigma_{ik}^2) = \frac{1}{\sqrt{2\pi}\sigma_{ik}} e^{-\frac{(x^{(i)} - \mu_{ik})^2}{2\sigma_{ik}^2}}$$

于是接下来的问题就在于如何估计 μ_{ik} 和 σ_{ik}^2 这两个参数了。为了便于叙述，我们把问题转化成这样的形式：假设现在有 N 个服从均值为 μ 、方差为 σ^2 的高斯分布的样本 x_1, x_2, \dots, x_N ，它们之间彼此独立，即

$$p(x_1, \dots, x_N | \mu, \sigma^2) = \prod_{i=1}^N p(x_i | \mu, \sigma^2)$$

现在我们要根据 $x_1 \sim x_N$ 的取值来算出 μ 和 σ^2 的 MLE，因此只需直接应用前文所说过的 MLE 公式即可：

$$\hat{\mu} = \arg \max_{\mu} \left[\sum_{i=1}^N \log p(x_i | \mu, \sigma^2) \right]$$

注意到

$$p(x_i | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i - \mu)^2}{2\sigma^2}}$$

所以就有

$$\begin{aligned}
\hat{\mu} &= \arg \max_{\mu} \left[\sum_{i=1}^N \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i - \mu)^2}{2\sigma^2}} \right] \\
&= \arg \max_{\mu} \left[\sum_{i=1}^N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{(x_i - \mu)^2}{2\sigma^2} \right] \\
&= - \arg \max_{\mu} \sum_{i=1}^N (x_i - \mu)^2
\end{aligned}$$

求导即得

$$-2 \sum_{i=1}^N (x_i - \hat{\mu}) = 0 \Rightarrow \hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i$$

即 μ 的 MLE 即为样本的经验均值。

类似的，可以计算出 $\hat{\sigma}^2$ 的 MLE（注意我们此时要用上 μ 的 MLE $\hat{\mu}$ ）：

$$\begin{aligned}
\hat{\sigma}^2 &= \arg \max_{\sigma} \left[\sum_{i=1}^N \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i - \hat{\mu})^2}{2\sigma^2}} \right] \\
&= \arg \max_{\sigma} \left[\sum_{i=1}^N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{(x_i - \hat{\mu})^2}{2\sigma^2} \right] \\
&= - \arg \max_{\sigma} \left[N \log \sigma + \sum_{i=1}^N \frac{1}{2\sigma^2} (x_i - \hat{\mu})^2 \right]
\end{aligned}$$

求导即得

$$-\frac{N}{\hat{\sigma}} + \sum_{i=1}^N \frac{(x_i - \hat{\mu})^2}{\hat{\sigma}^3} = 0 \Rightarrow \hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu})^2$$

即 σ^2 的 MLE 即为样本的经验方差。

至此，这个转化后的参数估计问题就被我们解决了，现在需要做的就是将该问题的解决方案迁移到朴素贝叶斯上去。这个迁移过程从直观上来说是很好的理解：对于每个特征 $x^{(i)}$ ，我们假设了当标签为 $y = c_k$ 时，有 $X^{(i)} \sim N(\mu_{ik}, \sigma_{ik}^2)$ ，那么 μ_{ik} 和 σ_{ik}^2 都用经验均值、方差去估计即可。具体而言，假设当 $y = c_k$ 时，特征 $x^{(i)}$ 取了如下 N_k 个值：

$$x_{k1}^{(i)}, x_{k2}^{(i)}, \dots, x_{kN_k}^{(i)}$$

那么直接估计

$$\hat{\mu}_{ik} = \frac{1}{N_k} \sum_{j=1}^{N_k} x_{kj}^{(i)}, \quad \hat{\sigma}_{ik}^2 = \frac{1}{N_k} \sum_{j=1}^{N_k} (x_{kj}^{(i)} - \hat{\mu}_{ik})^2$$

即可。于是与算法 2.1 类似的，我们可以写出当特征全是连续型特征时的朴素贝叶斯算法（我们一般会称算法 2.1 中的朴素贝叶斯为离散型朴素贝叶斯，称接下来的算法 2.2 中的朴素贝叶斯为连续型朴素贝叶斯）。

算法 2.2 连续型朴素贝叶斯算法

输入：训练数据集 $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$

过程：

(1) 估计先验概率 $p(y_k)$ ：

$$p(y_k) = \frac{N_k}{N}, k = 1, 2, \dots, K$$

(2) 计算“条件概率” $p(x^{(i)}|y_k; \theta)$ ：

$$p(x^{(i)}|y_k; \theta) = \frac{1}{\sqrt{2\pi}\hat{\sigma}_{ik}} e^{-\frac{(x^{(i)} - \hat{\mu}_{ik})^2}{2\hat{\sigma}_{ik}^2}}$$

这里有两个参数： $\hat{\mu}_{ik}$ 、 $\hat{\sigma}_{ik}^2$ ，它们可以用 MLE 定出：

$$\hat{\mu}_{ik} = \frac{1}{N_k} \sum_{j=1}^{N_k} x_{kj}^{(i)}$$

$$\hat{\sigma}_{ik}^2 = \frac{1}{N_k} \sum_{j=1}^{N_k} (x_{kj}^{(i)} - \hat{\mu}_{ik})^2$$

输出：朴素贝叶斯模型，能够估计数据 \mathbf{x}^* 的类别：

$$y^* = G(\mathbf{x}^*) = \arg \max_k p(y_k) \prod_{i=1}^n p(x^{*(i)}|y_k; \theta)$$

最后我们要考虑的，就是当特征向量中既有离散型特征又有连续型特征时该怎么办。不难看出，如果使用小区间切割法来直接离散化连续型特征的话，我们就能直接用算法 2.1 来建模。而如果想结合算法 2.1 和算法 2.2 的话，一种直观的做法就是将相应的公式直接合并在一起：

$$G(x) = \arg \max_k \left[\log p(y_k) + \sum_{x^{(i)} \in S_1} \log p(x^{(i)}|y_k; \theta) + \sum_{x^{(i)} \in S_2} \log p(x^{(i)}|y_k; \theta) \right]$$

其中， S_1 、 S_2 分别代表离散型特征集合和连续型特征集合，亦即

$$\begin{cases} p(x^{(i)}|y_k; \theta) = \frac{N_k^{(i)}}{N_k} & , \quad x^{(i)} \in S_1 \\ p(x^{(i)}|y_k; \theta) = \frac{1}{\sqrt{2\pi}\hat{\sigma}_{ik}} e^{-\frac{(x^{(i)} - \hat{\mu}_{ik})^2}{2\hat{\sigma}_{ik}^2}} & , \quad x^{(i)} \in S_2 \end{cases}$$

以上，我们对朴素贝叶斯算法的方方面面都做了大致的介绍。

2.2 决策树

2.1 节介绍的朴素贝叶斯算法虽然包含了不少数学理论和推导，但其模型的公式写出来却是非常简明的：

$$G(\mathbf{x}) = \arg \max_k p(y_k) \prod_{i=1}^n p(x^{(i)} | y_k; \theta)$$

而这一节将介绍的决策树算法则属于虽然公式写起来有很多，但其思想却是非常直观而容易理解的。尽管如此，基于它的许多算法（如 Random Forest、GBDT 及 xgboost）都是统计性能数一数二的模型，在现实任务中通常能表现得非常出色。

我们会先在 2.2.1 节简要介绍决策树的大致思路，然后用比较抽象的方式在 2.2.2 节和 2.2.3 节具体叙述决策树的两大算法，继而会在 2.2.4 节给出决策树的具体应用方法，并会在 2.2.5 节和 2.2.6 节补充一些实际操作层面的细节。在 2.2.7 节，我们会叙述如何应用决策树来做回归问题。

2.2.1 决策的方法

生活中总是充满着选择，人生的路也可以说是由数不清的决策所构成的。毫不夸张地说，一个人做决策的方式在很大程度上决定了他的成功与否。虽然人类在做决策时可能会用到各种奇怪的生理机制，但是对于机器而言，它的决策方式从本质上来说只有一种：就是根据输入的数据在经过一系列的计算后，根据计算结果来按照预先给定的指示或规则来进行决策。

不过，虽然和人类通常极具感情色彩的决策相比，机器决策背后的原理有些冷冰冰的感觉，但这并不妨碍我们让机器决策从形式上更贴合人类。决策树算法就是这样一种典型的算法，它认为我们做决策的过程通常是一个“决策路径”。比如，当我决定是否要辍学创业时，我可能会连续问自己三个问题：我是否有辍学的必要？创业的方向是否有前途？我是否有足够的资本来支撑创业？如果这三个问题我的回答都是“是”，那么我可能就会做出“辍学创业”的决策；反之，如果某一个问题的回答是“否”，那么我可能当即就会做出“好好学习”的决策，而不用继续问自己接下来的问题。

在上面这个例子中，“辍学是否必要→是否有前途→是否有资本”就是一个决策路径，只有当这个路径上的所有节点的回答都是“是”时我才会做出辍学创业的决策，否则在我回答“否”的节点上当机立断地做出继续学习的决策，而不会继续在这个决策路径上走下去。将这一整套逻辑翻译成图像的话，就可以得到如图 2.1 所示的模型（图片是使用 ProcessOn 在线绘制而成的）。

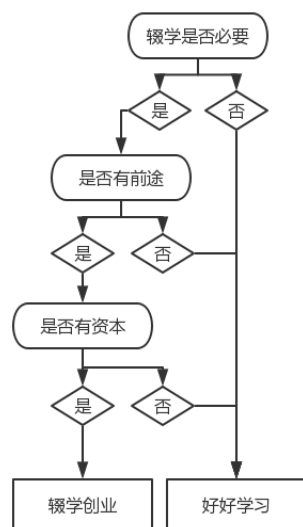


图2.1 是否辍学创业的决策路径

而事实上，图 2.1 所示的模型就已经是决策树模型了。我们的目标就是找到一种算法，使机器能够自动生成这样一个决策路径，从而生成类似图 2.1 所示的模型以帮助我们进行决策。如果用机器学习的术语来描述这个决策路径的话，不难看出：

- 我们可以将“辍学是否必要”“是否有前途”“是否有资本”这三个问题的答案看作三个特征，它们组成了样本中的特征向量。
- 我们可以将“辍学创业”和“好好学习”这两个决策看作样本中的标签。

于是就可以将我们的目的：根据三个问题的回答情况来决定是否辍学创业，看作根据特征向量来得到标签这样一个机器学习问题。更一般的，假设现在有一个 n 维的特征向量 $\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^T$ ，我们希望能根据它来做出决策 y ，在生成决策路径的这个思想下，可以很自然地得出这样一个算法：

- 挑选出 \mathbf{x} 中某个具有代表性的特征 $x^{(i)}$ 作为“问题”，根据它的取值（也就是“回答”）来分情况讨论。
 - 如果 $x^{(i)}$ 的某个取值足以让我们做出决策，就不再从该取值继续往下探索。
 - 如果 $x^{(i)}$ 的某个取值尚不足以让我们做出决策，那么就在该取值的基础上继续挑选出 \mathbf{x} 的另一个特征 $x^{(j)}$ 作为下一个“问题”，直到我们足以做出决策。

这其实就是决策树算法的直观说明了。不难看出，该算法有如下两个关键点：

- 怎样才算是“具有代表性的特征”？
- 怎样才算是“足以做出决策”？

正是由于在这两个关键点上可以有许许多多不同的做法，才导致了决策树算法的五花八门。我们将会在 2.2.2 节简要介绍这些做法的共性以及它们背后的抽象逻辑，且不失一般性，我们会暂时先只考虑分类问题。对于这些做法的具体差异和回归问题的解决方案，我们会分别在 2.2.5 节和 2.2.7 节中进行补充说明。

2.2.2 决策树的生成

本节我们将会介绍决策树算法是如何自动生成一个决策路径的。正如 2.2.1 节最后所说的，我们要解决的核心问题只有两个：定义“具有代表性的特征”和定义“足以做出决策”的基准。由于本书的主题是神经网络，所以我们在这一节将只会进行直观的叙述，相应的理论细节会放在最后的 2.2.5 节、2.2.6 节和 2.2.7 节中进行。

先来看怎么定义“具有代表性的特征”。从直观上来说，任何机器学习任务几乎总是可以归结为“发掘数据中的信息并以此降低对结果预测的不确定性”。换句话说，我们的目标几乎总是可以定为“不断地降低对结果预测的不确定性”这件事。那么衡量一个特征是否具有代表性的标准，就可以很自然地定义为“看它是否降低了结果预测的不确定性”。

于是现在问题转化为如何定义“不确定性”这个问题。由于机器接受的都是数字化的信息，所以对于机器而言，不确定性仅仅来源于信息的不足。所以我们可以通过定义“信息量”来作

为“不确定性”的相反数——拥有的信息越多则不确定性越小，拥有的信息越少则不确定性越大。

说到这里可能就已经有读者猜到，我们马上就要进入类似信息论这门学科知识的相关讨论了。仍然从直观上来看，当一个事件发生的概率很大时，我们通常认为它包含的信息很少，比如，我告诉你明天太阳依然会升起，你可能会觉得我说了句废话，毫无信息量可言；而当一个事件发生的概率很小时，我们通常认为它包含了大量的信息，比如，我告诉你明天太阳会从西边出来，你可能就会觉得这是一个大新闻，包含了巨额的信息量。正因此，我们在定义信息量（Information）时，通常会基于概率来定义：

$$H(p) \triangleq \text{information}, \quad p \in [0,1]$$

且我们会要求 H 随着 p 的增加而减少，即 H 应该是单调递减的。此外，由于我们要使用 H 来分析一个特征“是否具有代表性”，所以自然希望能够使 H 关于输入 p 是连续的，从而方便在理论上的分析。

基于上面两个基本假设，算法家提出了种种定义 H 的方法。我们将会在 2.2.5 节中对它们进行较为详细的介绍，这里就暂时先认为已经有了 H 这个能够帮助计算信息量的函数。

在有了 H 之后，选出“具有代表性的特征”，或者说“选出能够给予我们最大信息量的特征”，就是水到渠成的事情了。事实上，我们在 2.2.1 节中就已经说过，决策树算法可以直观地阐述为：

- 挑选出 \mathbf{x} 中的某个具有代表性的特征 $x^{(i)}$ 作为“问题”，根据它的取值（也就是“回答”）来分情况讨论。

这里面的“分情况讨论”如果用机器学习语言来说的话，就是“将当前的特征向量空间分成若干个子空间，然后在每个子空间里面进行讨论”。注意，由于我们的决策树是有明确的方向性的（因为它对应着一个决策路径），所以我们对特征向量空间的划分将会越来越细，且这些划分出来的子空间彼此之间不会相交。于是在机器学习的语言体系下，上面的阐述就可以说成：

- 挑选出 \mathbf{x} 中的某个具有代表性的特征 $x^{(i)}$ ，根据它来将当前特征向量空间 $\mathcal{X}_p^{(k)}$ 划分为若干个（比如说 q 个）互不相交的子空间 $\mathcal{X}_1^{(k+1)}, \mathcal{X}_2^{(k+1)}, \dots, \mathcal{X}_q^{(k+1)}$ ，然后在这些子空间中继续往下讨论。

所以如果我们想要选出能够给予我们最大信息量的特征的话，一个自然的想法就是选出这样一个 $x^{(i)}$ ，它在把 $\mathcal{X}_p^{(k)}$ 划分为 $\mathcal{X}_1^{(k+1)}, \mathcal{X}_2^{(k+1)}, \dots, \mathcal{X}_q^{(k+1)}$ 后，将会导致系统的信息量减少得最多。换句话说，假设 \tilde{H} 是能够计算特征向量空间的信息量的函数，那么我们就希望：

$$\Delta \tilde{H}(\mathcal{X}_p^{(k)}; x^{(i)}) \triangleq w_0(\mathcal{X}_p^{(k)}) \tilde{H}(\mathcal{X}_p^{(k)}) - \sum_{i=1}^q w_i(\mathcal{X}_i^{(k+1)}) \tilde{H}(\mathcal{X}_i^{(k+1)})$$

尽可能大，其中 w_i 是权重函数，表示对应空间的信息量的重要程度。这是因为 $\Delta \tilde{H}$ 越大就说明当

$x^{(i)}$ 被确定下来之后，系统整体减少的（加权）信息量就越大，从而我们就能认为 $x^{(i)}$ 所包含的信息量越大。

虽然从直观上来说似乎不太好定义一个特征向量空间的信息量是多少（即 $\tilde{H}(\mathcal{X}_p^{(k)})$ 似乎不太好定义），但考虑到这两条性质：

- 最根本的计算信息量的函数 H 接受的输入是一个概率。
- 最终的目的是想预测某个特征向量 \mathbf{x} 对应的标签 y 的概率。

$\tilde{H}(\mathcal{X}_p^{(k)})$ 的定义方法就一目了然了：假设特征向量空间 $\mathcal{X}_p^{(k)}$ 对应着标签空间 $\mathcal{Y}_p^{(k)}$ ，那么就可以直接将 $\tilde{H}(\mathcal{X}_p^{(k)})$ 定义为标签概率信息量的加权和。具体而言，假设我们一共有 K 个标签 c_1, c_2, \dots, c_K ，且事件 $y = c_i$ 简记为 y_i ，则

$$\tilde{H}(\mathcal{X}_p^{(k)}) \triangleq \sum_{i=1}^K w_i(y_i | \mathcal{Y}_p^{(k)}) H(p(y_i | \mathcal{Y}_p^{(k)}))$$

该式涉及一个权重函数 $w_i(y_i | \mathcal{Y}_p^{(k)})$ ，一个很自然的想法就是将它设置为相应标签的概率，即：

$$w_i(y_i | \mathcal{Y}_p^{(k)}) = p(y_i | \mathcal{Y}_p^{(k)})$$

于是就有

$$\tilde{H}(\mathcal{X}_p^{(k)}) \triangleq \sum_{i=1}^K p(y_i | \mathcal{Y}_p^{(k)}) H(p(y_i | \mathcal{Y}_p^{(k)}))$$

为了简单，我们把概率中的 $\mathcal{Y}_p^{(k)}$ 略去不写，那么上式就可以写成

$$\tilde{H}(\mathcal{X}_p^{(k)}) \triangleq \sum_{i=1}^K p(y_i) H(p(y_i))$$

不难看出，在 H 已经定义好的情况下， \tilde{H} 自然就已经被定义好了。而在 \tilde{H} 定义好的情况下，我们就能算出 $\Delta \tilde{H}(\mathcal{X}_p^{(k)}; x^{(i)})$ 的具体取值。此时，在特征向量空间 $\mathcal{X}_p^{(k)}$ 中“选出能够给予我们最大信息量的特征”就能很自然地翻译为“选出使得 $\Delta \tilde{H}(\mathcal{X}_p^{(k)}; x^{(i)})$ 最大的 $x^{(i)}$ ”，即 $\mathcal{X}_p^{(k)}$ 中的“最好的特征” $\hat{x}_p^{(k)}$ 应该满足

$$\hat{x}_p^{(k)} = \arg \max_{x^{(i)}} \Delta \tilde{H}(\mathcal{X}_p^{(k)}; x^{(i)})$$

至此，第一个关键点就解决了。下面我们来看看如何解决第二个关键点，即如何定义“足以做出决策”这件事。事实上，在解决第一个关键点的过程中，我们已经或多或少地把第二个关键点包含进去了——从直观上来看，我们完全可以认为“足以做出决策”当且仅当此时的特征向量空间的信息量很少。这是因为信息量很少就意味着当前子空间内各个事件的发生与否已经基本没有随机性，所以当然就可以做出决策了。用数学语言来表达的话，就是当 $\tilde{H}(\mathcal{X}_p^{(k)}) < \epsilon$

时,我们认为 $\mathcal{X}_p^{(k)}$ 足以做出决策,其中 ϵ 是我们预先设定好的一个超参数。

那么在 $\mathcal{X}_p^{(k)}$ “足以做出决策”后,我们应该怎样去做决策呢?答案也是很直观的:只需把 $\mathcal{X}_p^{(k)}$ 对应的标签空间 $\mathcal{Y}_p^{(k)}$ 中出现概率最大的 $\hat{y}_p^{(k)}$ 作为整个特征向量空间 $\mathcal{X}_p^{(k)}$ 的标签即可。换句话说,对于一个新的特征向量 \mathbf{x} ,如果它属于 $\mathcal{X}_p^{(k)}$,那么模型 G 的输出就应该是 $\hat{y}_p^{(k)}$,即:

$$\mathbf{x} \in \mathcal{X}_p^{(k)} \Rightarrow G(\mathbf{x}) = \hat{y}_p^{(k)} = \arg \max_{y_p^{(k)}} p(y_p^{(k)} | \mathcal{Y}_p^{(k)})$$

至此,第二个关键点也已解决。只要把这两个关键点的解决方案结合起来,我们就能够写出决策树的生成算法了。不过在此之前,让我们先来进行一个小结以加深对整体的认知。具体而言,从上述的讨论中,我们可以归纳出决策树的两条核心性质:

- 决策树的生成过程从本质上来说,其实是不断地挑选出特征并对特征向量空间进行划分的过程。当空间被划分到信息量只剩很少时,我们就会给这个子空间打上一个标签。于是当一个新的特征向量进入模型时,我们只需看它属于哪个子空间,然后把相应的标签输出即可。
- 由于决策树的生成具有明确的方向性(因为它对应着一条决策路径),所以不难想象,这些被划分出来的子空间彼此之间是互不相交的,即对于任意一个新的特征向量而言,它属于且仅属于其中某个子空间。这就避免了一个特征向量同属两个子空间而这两个子空间的标签不一致的问题。

注意:“划分”这两个字其实是一个数学术语:我们称 X_1, X_2, \dots, X_n 是 X 的一个划分,当且仅当 $X = X_1 \cup X_2 \cup \dots \cup X_n$ 且 $X_1 \sim X_n$ 两两之间的交为空。可以看到,我们上面所说的划分是满足这两个要求的。为了简单,我们在后文中如果用到了划分的话,如果没有特别指出,默认是这种“并起来能组成全体,两两之间不会相交”的划分。

在这两条核心性质的指导下,我们就能比较自然地写出(抽象的)决策树的生成算法了,如算法2.3所示。

算法2.3 决策树生成的(抽象)算法

输入: 特征向量空间 \mathcal{X} , 标签空间 \mathcal{Y} 给定的阈值 ϵ

过程:

(1) 初始化当前特征向量空间 $\tilde{\mathcal{X}}$ 和对应的标签空间 $\tilde{\mathcal{Y}}$ 为 \mathcal{X} 、 \mathcal{Y}

(2) 看看 $\tilde{\mathcal{X}}$ 是否足以做出决策,即看看是否有

$$\tilde{H}(\tilde{\mathcal{X}}) < \epsilon$$

(3) 若是,则给 $\tilde{\mathcal{X}}$ 打上标签 \tilde{y} ,其中 \tilde{y} 满足

$$\tilde{y} = \arg \max_y p(y | \tilde{\mathcal{Y}})$$

(4) 否则,选出能给我们带来最大信息量的特征 \tilde{x} :

$$\tilde{x} = \arg \max_x \Delta \tilde{H}(\tilde{\mathcal{X}}; \mathbf{x})$$

(5) 根据 \tilde{x} 将我们的 $\tilde{\mathcal{X}}$ 划分成若干个子空间,对这些子空间运行从(2)开始的算法,直至所有子空间都被打上了标签
输出: 一个特征向量空间 \mathcal{X} 的划分,且这些划分都被打上了标签

由该算法不难看出，决策树的生成确实具有明确的方向性。也正因此，我们可以把算法中的第（2）～（5）步看成是决策树中的“一层”。换句话说，可以把初始的特征向量空间 \mathcal{X} 看成是“第0层”，然后接下来每次划分出来的子空间的总体看成是新的“一层”。具体而言，假设在第1次划分中， \mathcal{X} 被划分为 $m^{(1)}$ 个子空间：

$$\mathcal{X} \rightarrow \{\mathcal{X}_1^{(1)}, \dots, \mathcal{X}_{m^{(1)}}^{(1)}\}$$

那么我们会把

$$\{\mathcal{X}_1^{(1)}, \dots, \mathcal{X}_{m^{(1)}}^{(1)}\}$$

看成第1层。继而假设继续把第1层中的第 i 个子空间划分为了 $m^{(2)(i)}$ 个子空间（如果第 i 个子空间还不足以做出决策的话）：

$$\mathcal{X}_i^{(1)} \rightarrow \{\mathcal{X}_1^{(2)(i)}, \dots, \mathcal{X}_{m^{(2)(i)}}^{(2)(i)}\}, \quad i = 1, 2, \dots, m^{(1)}$$

我们就会把

$$\{\mathcal{X}_1^{(2)(1)}, \dots, \mathcal{X}_{m^{(2)(1)}}^{(2)(1)}, \mathcal{X}_1^{(2)(2)}, \dots, \mathcal{X}_{m^{(2)(2)}}^{(2)(2)}, \dots, \mathcal{X}_1^{(2)(m^{(1)})}, \dots, \mathcal{X}_{m^{(2)(m^{(1)})}}^{(2)(m^{(1)})}\}$$

看成第2层。为了书写方便，令

$$m^{(2)} \triangleq \sum_{i=1}^{m^{(1)}} m^{(2)(i)}$$

我们就能把第2层写成

$$\{\mathcal{X}_1^{(2)}, \dots, \mathcal{X}_{m^{(2)}}^{(2)}\}$$

更一般的，假设在接下来的第 k 次划分中，我们把来自第 $k-1$ 层中的子空间 $\mathcal{X}_i^{(k-1)}$ 划分成 $m^{(k)(i)}$ 个子空间：

$$\mathcal{X}_i^{(k-1)} \rightarrow \{\mathcal{X}_1^{(k)(i)}, \dots, \mathcal{X}_{m^{(k)(i)}}^{(k)(i)}\}, \quad i = 1, 2, \dots, m^{(k-1)}$$

我们就会把

$$\{\mathcal{X}_1^{(k)(1)}, \dots, \mathcal{X}_{m^{(k)(1)}}^{(k)(1)}, \mathcal{X}_1^{(k)(2)}, \dots, \mathcal{X}_{m^{(k)(2)}}^{(k)(2)}, \dots, \mathcal{X}_1^{(k)(m^{(k-1)})}, \dots, \mathcal{X}_{m^{(k)(m^{(k-1)})}}^{(k)(m^{(k-1)})}\}$$

看成第 k 层。为了书写方便，令

$$m^{(k)} \triangleq \sum_{i=1}^{m^{(k-1)}} m^{(k)(i)}$$

我们就能把第 k 层写成

$$\{x_1^{(k)}, \dots, x_{m^{(k)}}^{(k)}\}$$

把上面这些叙述可视化的话，就能得到如图 2.2 所示的决策树结构。

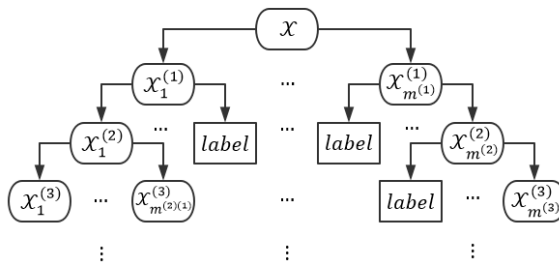


图2.2 (抽象) 决策树结构的示意图

其中，每个圆角矩阵都是“还不足以做出决策”的子空间，我们需要对它们进行进一步的划分；每个方角矩阵则都是“足以做出决策”的子空间，我们需要给它们打上标签 (label)。

此外，由于我们在实际任务中只能获得一个数据集而无法获得整个空间，所以在实际应用中，算法 2.3 这种抽象算法必定要做出修改，不过背后的思想和逻辑是相通的。相关的讨论会放在 2.2.5 节中进行，有兴趣的读者可以先行参见相应的部分。

2.2.3 决策树的剪枝

在第 1 章中，我们曾经说过“过拟合”这个概念：如果 G 在训练数据 D 上表现优异，但在 \mathcal{D} 的其余采样（比如说测试集）上表现糟糕，那么我们就说产生了过拟合的情况。在决策树中，这个问题体现得尤为明显，因为不难想象，在我们只能观察到训练数据集 D 的前提下，只要把特征向量空间 \mathcal{X} 划分得足够细，那么模型 G 在 D 上的表现就能达到完美。一个极端的例子是，假设

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

那么我们只需将 \mathcal{X} 划分成 $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_N$ ，使得

$$x_i \in \mathcal{X}_i, \quad \forall i = 1, 2, \dots, N$$

然后再把标签 y_i 打在 \mathcal{X}_i 上即可。不过可想而知的是，除非 D 真的是 \mathcal{D} 的一个非常完美的采样，否则 G 的泛化能力肯定会很差。

所以在把决策树生成出来后，还需要做出一些措施来防止它过拟合。我们在第 1 章中还说过，从实际操作上来说，防止过拟合的非常普遍的做法就是对 G 的复杂度做出一定的惩罚，从而使 G 趋于精简。对于决策树来说，一个非常直观的做法就是“剪枝”——把一些多余的枝叶给“剪掉”，从而让决策树的复杂度降低。以图 2.2 所示的决策树结构为例，所谓的剪枝，就是把图中某些圆角矩阵下面的分支都“剪掉”，从而把它转化为一个方角矩阵。

虽然像这种“先生成，后剪枝”的做法非常常见，但我们也可以通过其他手段来防止过拟

合。比如，最直观的一种做法就是把算法 2.3 中的阈值 ϵ 调大一点，这样一来决策树就会倾向于认为当前子空间已经足以做出决策，从而防止决策树“过度生成”并趋于简单。我们通常把“在决策树的生成过程中加入比较宽松的停止条件”这种做法称为“预剪枝（Pre-Pruning）”，并把“先让决策树充分生成，然后再进行剪枝”这种做法称为“后剪枝（Post-Pruning）”。

由前文的讨论可知，后剪枝的关键在于选出一个圆角矩阵并将它转化为方角矩阵，我们通常称该过程为“局部剪枝”，从而后剪枝的关键就转化成如何定出局部剪枝的标准。通常来说有两种做法：

- 应用交叉验证的思想，若局部剪枝能够使模型在测试集上的错误率降低，则进行相应的局部剪枝（预剪枝中也应用了类似的思想）。
- 应用正则化的思想，综合考虑信息量多少和模型复杂度并定出一个函数 H ，用该函数的函数值来衡量是否应该对一个圆角矩阵进行局部剪枝。

然而遗憾的是，虽然 `scikit-learn` 允许我们进行预剪枝，但它却不支持进行后剪枝。不过不必担心，我们会在 4.3.4 节中介绍一种利用神经网络对决策树进行类似于后剪枝操作的方法。

2.2.4 实例演示

我们仍以 2.1 节介绍朴素贝叶斯时所用到的蘑菇数据集为例，应用决策树解决该问题的代码实现如下所示：

```
01 from sklearn.tree import DecisionTreeClassifier
02
03 clf = DecisionTreeClassifier()
04 clf.fit(x_train, y_train)
05 print(np.mean(y_test == clf.predict(x_test)))
```

注意：虽然 `scikit-learn` 的 `DecisionTreeClassifier` 不要求输入数据是 OneHot Encoding 的数据，但是只有在使用 OneHot Encoding 的输入数据时，`DecisionTreeClassifier` 的表现才是最合理的。其中的细节我们会在 2.2.6 节里进行说明，有需要的读者可以参阅相应部分。

得到的结果是 0.999058380414，即准确率在 99.91% 左右，可以说是一个非常好的结果。不过由于这结果实在太好了，所以可能已经有读者心生疑虑：为什么能够有这么好的效果？这就需要对数据进行适当的挖掘。我们会在附录 D 中介绍各种模型的可视化时给出决策树在蘑菇数据集上的表现如此优异的解释，读者将会看到蘑菇数据集中有一个特征至关重要，而决策树算法的性质使得决策树能够轻易地把这个特征挑出来。

2.2.5* 决策树的三大算法

在 2.2.2 节中，我们在认为已经有了 H 这个能够计算信息量的函数之后，推导出了算法 2.3 这个较为抽象的算法。在这一节中，我们先介绍两种 H 的定义方法，然后把算法 2.3 写成三种

能够实际应用的形式。

回忆前文关于 H 的讨论，我们知道它需要满足这样两个性质：

- H 接受的输入 p 是一个概率值，即 $p \in [0,1]$ 。
- H 是关于 p 的连续单调递减函数。

那么考虑到简单性，我们就能比较自然地想到两种类型的 H 。具体而言：

- H 关于 p 是线性的，此时可以令

$$H(p) = 1 - p$$

- H 关于 p 是非线性的，此时可以令

$$H(p) = -\log p$$

注意：这两种定义方式并不是拍脑袋想出来的，它们背后有着相当深厚的数学理论背景（比如对称性、可加性等）。不过由于相关讨论超出了本书的范围，所以就不展开说明了。

在有了 H 之后，根据 $\tilde{H}(\mathcal{X}) = \sum_{i=1}^K p(y_i)H(p(y_i))$ ，我们就能直接写出相应的 \tilde{H} 了：

- H 关于 p 是线性的，则

$$\tilde{H}(\mathcal{X}) = \sum_{i=1}^K p(y_i)(1 - p(y_i)) = \sum_{i=1}^K p(y_i) - \sum_{i=1}^K p^2(y_i) = 1 - \sum_{i=1}^K p^2(y_i)$$

我们通常会称此时的 \tilde{H} 为基尼系数（Gini Index）。

- H 关于 p 是非线性的，则

$$\tilde{H}(\mathcal{X}) = \sum_{i=1}^K p(y_i) \cdot -\log p(y_i) = -\sum_{i=1}^K p(y_i) \log p(y_i)$$

我们通常会称此时的 \tilde{H} 为信息熵（Entropy）。

到此为止，我们讨论的都还是比较抽象的东西。当我们想要实际应用它们时，需要做一些估计，而一个直观、合理的估计就是拿数据集去估计相应的空间。换句话说，我们只需把上面说过的和空间有关的公式，全部用数据集来重写一遍即可。事实上，回忆 2.1 节介绍的朴素贝叶斯算法，我们曾经说过，标签的先验概率的 MLE 即为频率估计概率的结果，即：

$$\hat{p}(y_i)_{MLE} = \frac{N_i}{N}$$

在决策树算法中，我们使用的仍然是这种估计。所以在实际操作上，如果想要计算某个特征向量空间 \mathcal{X} 的信息量，我们会先对 \mathcal{X} 所属的样本空间 \mathcal{D} 采样出一个样本 D ，其中

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$$

然后再在 D 上面计算出 \tilde{H} 的值并拿它作为信息量的估计：

$$\tilde{H}(\mathcal{X}) \approx \tilde{H}(D) = \begin{cases} 1 - \sum_{i=1}^K \left(\frac{N_i}{N}\right)^2, & \text{if } \tilde{H} \text{ is Gini Index} \\ - \sum_{i=1}^K \frac{N_i}{N} \log \frac{N_i}{N}, & \text{if } \tilde{H} \text{ is Entropy} \end{cases}$$

可以看到，这就是用数据集 (D) 去估计空间 (\mathcal{X}) 的典型例子。

需要注意的是，当 \tilde{H} 是 Entropy 时，会涉及对数的底应该取多少的问题。通常来说有两种取法：

- 将对数的底取为 2，此时 Entropy 的单位是比特 (bit)。
- 将对数的底取为 e (即取自然对数)，此时 Entropy 的单位是纳特 (nat)。

在知道如何计算出 \tilde{H} 之后，假设当前的某个特征 $x^{(i)}$ 能把当前的特征向量空间 \mathcal{X} 划分为 q 个子空间 (如何具体进行划分会在 2.2.6 节中讨论)：

$$\mathcal{X} \xrightarrow{x^{(i)}} \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_q\}$$

那么根据 \tilde{H} 的公式，就能算出 $x^{(i)}$ 能给我们带来的信息量：

$$\Delta \tilde{H}(\mathcal{X}; x^{(i)}) = w_0(\mathcal{X}) \tilde{H}(\mathcal{X}) - \sum_{i=1}^q w_i(\mathcal{X}_i) \tilde{H}(\mathcal{X}_i)$$

由前文的讨论可知，决策树生成的关键就在于选出最好的、能给我们带来最大信息量的特征 \hat{x} 。具体而言，就是

$$\hat{x} = \arg \max_{x^{(i)}} \Delta \tilde{H}(\mathcal{X}; x^{(i)})$$

与计算 H 类似，在实际计算 $\Delta \tilde{H}$ 时，我们会采用已有的数据来进行估计。换句话说，在实际任务中，我们所做的假设是：

$$D \xrightarrow{x^{(i)}} \{D_1, D_2, \dots, D_q\}$$

从而

$$\hat{x} = \arg \max_{x^{(i)}} \Delta \tilde{H}(D; x^{(i)})$$

其中 D 是当前所拥有的数据集。

在知道如何从数据集出发找到最好的特征之后，我们就要来看如何从数据集出发来做出决策了。这一步是非常简单的，假设当前的数据集为

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$$

那么只需选出 y_1, y_2, \dots, y_N 中出现最多的标签来作为对应的样本向量空间的标签即可。用数学语言来表达的话，就是若

$$\tilde{y} = \arg \max_{c_k} \sum_{i=1}^N I(y_i = c_k)$$

那么我们就给 D 对应的特征向量空间 \mathcal{X} 打上标签 \tilde{y} 。注意，这里我们用到了“示性函数” I ，它的定义如下：

$$I(x) = \begin{cases} 1 & , \quad \text{if } x \text{ is true} \\ 0 & , \quad \text{if } x \text{ is not true} \end{cases}$$

换句话说，当 I 的输入为真时， I 就会输出 1，反之就会输出 0。故 $\sum_{i=1}^N I(y_i = c_k)$ 所表达的，其实就是“ $y_1 \sim y_N$ 中值为 c_k 的个数”。

综上所述，我们就能总结出算法 2.4 所示的一套可实际操作的算法。

算法 2.4 决策树的生成算法

输入：训练数据集 D 和给定的阈值 ϵ

过程：

(1) 初始化当前数据集为 D

(2) 看看 D 中的样本是否足以做出决策，即看看是否有

$$\tilde{H}(D) < \epsilon$$

(3) 若是，则给 D 所对应的特征向量空间 \mathcal{X} 打上标签 \tilde{y} ，其中 \tilde{y} 满足

$$\tilde{y} = \arg \max_{c_k} \sum_i I(y_i = c_k)$$

(4) 否则，选出能给我们带来最大信息量的特征 \tilde{x} ：

$$\tilde{x} = \arg \max_{x^{(i)}} \Delta \tilde{H}(D; x^{(i)})$$

(5) 根据 \tilde{x} 来将我们的 D 划分成若干个子数据集，对这些子数据集运行从步骤 (2) 开始的算法，直至所有子数据对应的子空间都被打上了标签

输出：一个数据集 D 的划分，且这些划分所对应的特征向量空间 \mathcal{X} 都被打上了标签

所以剩下下来的问题其实是如何具体计算 $\Delta \tilde{H}$ 了。注意，

$$\Delta \tilde{H}(D; x^{(i)}) = w_0(D) \tilde{H}(D) - \sum_{j=1}^q w_j(D_j) \tilde{H}(D_j)$$

所以核心问题就又转化为如何选择 \tilde{H} （选 Gini Index 还是 Entropy）以及如何定义权重函数 $w_j(D_j)$ 。下面我们就来介绍一下决策树中的三大算法——ID3、C4.5 和 CART 分别是如何选择、定义 \tilde{H} 和 w_j 的。

1. ID3 (Interactive Dichotomizer-3, 交互式二分法)

在 ID3 算法中, 我们选择了 Entropy 作为 \tilde{H} , 同时定义 w_i 的方式也很简单直观:

$$w_0^{ID3}(D) \triangleq 1, \quad w_j^{ID3}(D_j) \triangleq \frac{|D_j|}{|D|}$$

其中, $|D_j|$ 、 $|D|$ 分别代表 D_j 、 D 中的样本数。于是 $\Delta\tilde{H}$ 的公式即为

$$\begin{aligned} \Delta\tilde{H}^{ID3}(D; x^{(i)}) &= w_0^{ID3}(D)\tilde{H}(D) - \sum_{j=1}^q w_j^{ID3}(D_j)\tilde{H}(D_j) \\ &= -\sum_{i=1}^K \frac{N_i}{N} \log \frac{N_i}{N} + \sum_{j=1}^q \frac{|D_j|}{|D|} \sum_{i=1}^K \frac{N_{ji}}{N_j} \log \frac{N_{ji}}{N_j} \end{aligned}$$

其中, N_j 为子数据集 D_j 的样本数 (即 $N_j = |D_j|$), N_{ji} 为 D_j 中属于第 i 类的样本数。

像 ID3 这样定义的 $\Delta\tilde{H}^{ID3}$, 我们通常称之为互信息 (Mutual Information)。我们可以证明互信息一定是非负的, 即

$$\Delta\tilde{H}^{ID3}(D; x^{(i)}) \geq 0, \quad \forall D, x^{(i)}$$

此外, 互信息存在着这样一个特性: 如果某一个特征可以把当前数据集 D 分成很多份的话, 我们会偏向于去选择它。换句话说, 我们会倾向于选择能使 q 比较大的特征。这主要是因为, 从极端的情况来看, 假设一个特征 x 能直接把 D 分成 N 个子数据集, 且每个子数据集都包含且仅包含一个样本的话, 那么划分后的信息量必然是 0 (因为不再有不不确定性), 从而互信息就会认为 x 是最好的特征。

然而这个特性却不是一个好的特性。正如前文所说, 此时模型的泛化能力必然会很差, 即 x 事实上并不是一个好的特征。所以我们需要对互信息做出一些改进, 比如, 对 q 做出一个惩罚。在这种思想下, 我们就能得到 C4.5 算法。

2. C4.5

和 ID3 算法相比, C4.5 对权重做了一个放缩:

$$w_j^{C4.5} \triangleq \frac{w_j^{ID3}}{\hat{H}(x^{(i)})}, \quad j = 0, 1, \dots, q$$

这其实等价于对整个 $\Delta\tilde{H}$ 做了一个放缩:

$$\Delta\tilde{H}^{C4.5} = \frac{\Delta\tilde{H}^{ID3}}{\hat{H}(x^{(i)})}$$

其中

$$\hat{H}(x^{(i)}) \triangleq -\sum_{j=1}^q \frac{|D_j|}{|D|} \log \frac{|D_j|}{|D|}$$

该定义式和 Entropy 的形式一致，所以它通常被称为“特征 $x^{(i)}$ 的 Entropy”。从直观上不难看出，当 q 变大时，由于特征 $x^{(i)}$ 能把系统划分得更细致，从而能使划分后的系统信息量变小，所以特征 $x^{(i)}$ 本身的信息量就会变大，即 $\hat{H}(x^{(i)})$ 就会变大。注意，我们曾经说过，互信息 $\Delta\tilde{H}^{ID3}(D; x^{(i)})$ 是非负的，所以此时 C4.5 下的 $\Delta\tilde{H}$ ：

$$\Delta\tilde{H}^{C4.5}(D; x^{(i)}) = \frac{1}{\hat{H}(x^{(i)})} \cdot \Delta\tilde{H}^{ID3}(D; x^{(i)})$$

就会变小，这就达到了对 q 做出惩罚的目的。

我们通常会称 $\Delta\tilde{H}^{C4.5}(D; x^{(i)})$ 为信息增益比 (Information Gain Ratio)，它是用得比较多的、比单纯的互信息在一般情况下更为合理的度量。

注意：C4.5 算法虽然不会倾向于选择 q 比较大的特征，但却有可能倾向于选择 q 比较小的特征。针对这个问题，Quinlan 在 1993 年提出了一个启发式的方法：先选出互信息比平均互信息要高的特征，然后从这些特征中选出信息增益比最高的。

3. CART (Classification and Regression Tree, 分类与回归树)

和 ID3 算法相比，CART 选择了 Gini Index 作为 \tilde{H} ，权重的定义则没变：

$$\begin{aligned}\Delta\tilde{H}^{CART}(D; x^{(i)}) &= \left[1 - \sum_{i=1}^K \left(\frac{N_i}{N} \right)^2 \right] - \sum_{j=1}^q \frac{|D_j|}{|D|} \left[1 - \sum_{i=1}^K \left(\frac{N_{ji}}{N_j} \right)^2 \right] \\ &= 1 - \sum_{j=1}^q \frac{|D_j|}{|D|} - \sum_{i=1}^K \left(\frac{N_i}{N} \right)^2 + \sum_{j=1}^q \frac{|D_j|}{|D|} \sum_{i=1}^K \left(\frac{N_{ji}}{N_j} \right)^2 \\ &= - \sum_{i=1}^K \left(\frac{N_i}{N} \right)^2 + \sum_{j=1}^q \frac{|D_j|}{|D|} \sum_{i=1}^K \left(\frac{N_{ji}}{N_j} \right)^2\end{aligned}$$

2.2.6* 数据集的划分

我们此前一直没有说明如何具体地进行“利用特征向量 \mathbf{x} 中的某个特征 $x^{(i)}$ 将当前数据集 D 划分成 q 个子数据集”这步操作，这一节我们将对它进行比较完善的补充说明。

首先来考虑特征 $x^{(i)}$ 是离散型特征的情况，假设它有 $n^{(i)}$ 个取值：

$$x^{(i)} \in \{x_1^{(i)}, x_2^{(i)}, \dots, x_{n^{(i)}}^{(i)}\}$$

那么一个自然的想法就是将数据集 D 划分成 $n^{(i)}$ 个子数据集 $D_1, D_2, \dots, D_{n^{(i)}}$ ，其中第 j 个子数据集 D_j 中的所有样本的特征 $x^{(i)}$ 的取值都需要是 $x_j^{(i)}$ ，即

$$D \xrightarrow{x^{(i)}} \{D_1, D_2, \dots, D_{n^{(i)}}\}$$

其中

$$D_j = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \text{ and } x^{(i)} = x_j^{(i)}\}, \quad \forall j = 1, 2, \dots, n^{(i)}$$

在数学符号中，我们会用符号“ \wedge ”来表示“and”，从而上式就可以写成

$$D_j = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} = x_j^{(i)}\}, \quad \forall j = 1, 2, \dots, n^{(i)}$$

而如果特征 $x^{(i)}$ 是连续型特征的话，同样的思路就不管用了。因为虽然在给定的数据集 D 下， $x^{(i)}$ 确实可能只有 $n^{(i)}$ 个取值，但是在对应的特征向量空间中来看的话， $x^{(i)}$ 的取值个数就很有可能是无限的了。此时一种常见的做法是，选定一个阈值 $\tilde{\epsilon}^{(i)}$ ，构造出一个新的特征 $\tilde{x}^{(i)}$ ，它是“特征 $x^{(i)}$ 的值不大于阈值 $\tilde{\epsilon}^{(i)}$ ”的示性函数：

$$\tilde{x}^{(i)} = I(x^{(i)} \leq \tilde{\epsilon}^{(i)}) = \begin{cases} 1 & , \quad x^{(i)} \leq \tilde{\epsilon}^{(i)} \\ 0 & , \quad x^{(i)} > \tilde{\epsilon}^{(i)} \end{cases}$$

换句话说，就是

$$D \xrightarrow{\tilde{x}^{(i)}} \{D_1, D_2\}$$

其中

$$D_1 = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} \leq \tilde{\epsilon}^{(i)}\}$$

$$D_2 = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} > \tilde{\epsilon}^{(i)}\}$$

不难看出，阈值 $\tilde{\epsilon}^{(i)}$ 的选取是这种划分方式是否合理的关键所在。一个比较容易想到的选取方法是：

- 假设 $x^{(i)}$ 在数据集 D 中的 $n^{(i)}$ 个取值分别是 $x_1^{(i)}, x_2^{(i)}, \dots, x_{n^{(i)}}^{(i)}$ 。不失一般性，再不妨假设它们满足 $x_1^{(i)} < x_2^{(i)} < \dots < x_{n^{(i)}}^{(i)}$ （若不然，进行一次排序操作即可），那么就可以简单地依次选择 $\tilde{\epsilon}_1^{(i)}, \tilde{\epsilon}_2^{(i)}, \dots, \tilde{\epsilon}_{m^{(i)}}^{(i)}$ 作为阈值并决出最好的一个，其中我们要求 $\tilde{\epsilon}_1^{(i)} \sim \tilde{\epsilon}_{m^{(i)}}^{(i)}$ 构成等差数列，且：

$$\tilde{\epsilon}_1^{(i)} = x_1^{(i)}, \quad \tilde{\epsilon}_{m^{(i)}}^{(i)} = x_{n^{(i)}}^{(i)}$$

$m^{(i)}$ 的选取则视情况而定。一般而言会取 $m^{(i)}$ 反比于“深度”。这意味着当数据越分越细时，对特征的划分就会越来越粗糙，从直观上来说这有益于防止过拟合。

不过这种做法存在一个致命的缺陷：它可能会产生非常多“冗余”的阈值。试想如果这些取值满足：

$$x_1^{(i)} = 0, x_2^{(i)} = 1000, x_3^{(i)} = 1001, \dots$$

那么我们就会在 $x_1^{(i)}$ 和 $x_2^{(i)}$ 之间尝试大量的划分标准，但显然这些划分标准算出来的结果都

是一样的。为了处理类似于这种不合理的情况，我们可以做出如下改进：

- 依次选择

$$\tilde{\epsilon}_1^{(i)} = \frac{x_1^{(i)} + x_2^{(i)}}{2}, \dots, \tilde{\epsilon}_{m^{(i)}}^{(i)} = \frac{x_{n^{(i)}-1}^{(i)} + x_{n^{(i)}}^{(i)}}{2}$$

作为阈值并决出最好的一个。不难看出，此时 $m^{(i)} = n^{(i)} - 1$ 。

在这种做法的基础之上，还有一种理论上能够得到加速的做法：

- 设 $x_1^{(i)}, x_2^{(i)}, \dots, x_{n^{(i)}}^{(i)}$ 所对应的标签分别是 $y_1^{(i)}, \dots, y_{n^{(i)}}^{(i)}$ ，那么我们就只会在 $\tilde{\epsilon}_1^{(i)} \sim \tilde{\epsilon}_{m^{(i)}}^{(i)}$ 中选取，使得：

$$y_j^{(i)} \neq y_{j+1}^{(i)}, \quad (i = 1, \dots, m - 1)$$

的 $\tilde{\epsilon}_j^{(i)}$ 作为阈值并决出最好的一个。

这种做法在某些情况下会表现得更好，但在某些情况下会显得不合理。

以上我们就完成了理论上的讨论，不过如果考虑到实际实现的话，会发现尚有瑕疵之处：当特征 $x^{(i)}$ 的类型不同时，划分的行为就可能大相径庭（为离散型时有可能划分出很多个，为连续型时只可能划分出两个）。所以一个很自然的想法就是，能否想方设法让 $x^{(i)}$ 是离散型特征时，划分出来的子数据集也只有两个。回忆在 2.1.6 节中讨论朴素贝叶斯算法时我们曾经说过，当对输入数据完成 OneHot Encoding 后，我们就能把 $x^{(i)}$ 展开成 $\tilde{x}^{(i)}$ ，即将一个多值特征展开成若干个二值特征，而在决策树中我们也可以这么做。具体而言，不妨假设特征向量 \mathbf{x} 中的离散型特征为 $x^{(1)}, x^{(2)}, \dots, x^{(m)}$ ，我们现在的做法是：

$$\hat{x} = \arg \max_{x^{(i)}} \Delta \tilde{H}(D; x^{(i)})$$

而在完成 OneHot Encoding 后，假设 $x^{(i)}$ 的 $n^{(i)}$ 个取值为 $x^{(i)(1)}, x^{(i)(2)}, \dots, x^{(i)(n^{(i)})}$ ，那么我们就很自然地把 $\Delta \tilde{H}(D; x^{(i)})$ 展开成 $n^{(i)}$ 个 $\Delta \tilde{H}(D; \tilde{x}^{(i)(j)})$ ，其中

$$\tilde{x}^{(i)(j)} = \begin{cases} 1 & , \text{ if } x^{(i)} = x^{(i)(j)} \\ 0 & , \text{ if } x^{(i)} \neq x^{(i)(j)} \end{cases}$$

从而就有

$$\hat{x} = \arg \max_{\tilde{x}^{(i)(j)}} \Delta \tilde{H}(D; \tilde{x}^{(i)(j)})$$

综上所述，可以发现无论特征向量 \mathbf{x} 中每个特征 $x^{(i)}$ 的类型是什么，我们总是能把选出最优特征的方程统一写成

$$\hat{x} = \arg \max_{\tilde{x}^{(i)}} \Delta \tilde{H}(D; \tilde{x}^{(i)})$$

其中每个 $\tilde{x}^{(i)}$ 都是只取 0 或 1 的二值特征。不难想象，在将离散型特征和连续型特征都统一

为二值特征后，我们的决策树在每次“回答问题”时都只会回答“是”或“否”（分别对应着取值 1 和取值 0），从而最终我们得到的决策树就会是一棵二叉树。

讲到这里，我们就能回答 2.2.4 节中遗留下来的一个问题了：为什么只有在使用经过了 OneHot Encoding 的数据作为输入时，scikit-learn 的 DecisionTreeClassifier 的表现才是最为合理的？这是因为在 scikit-learn 中，它把所有的特征都视为连续型特征。即使特征 $x^{(i)}$ 是离散型特征，它也会尝试找到阈值 $\epsilon^{(i)}$ ，并认为划分后的两个子数据集是

$$D_1 = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} \leq \epsilon^{(i)}\}$$

$$D_2 = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} > \epsilon^{(i)}\}$$

比如，假设 $x^{(i)}$ 有 4 个取值：0、1、2 和 3 并假设 scikit-learn 找到的阈值是 1.5，那么划分后的两个子数据集就是

$$D_1 = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} \in \{0, 1\}\}$$

$$D_2 = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in D \wedge x^{(i)} \in \{2, 3\}\}$$

这当然是不太合理的，因为离散型特征 $x^{(i)}$ 的各个取值之间的关系和它们在数值上的取值本来是应该毫无关系的，即不能说取值为 0 和取值为 1 的关系要比取值为 0 和取值为 3 的关系要紧密；然而这种强行找阈值的方法却把连在一起的取值视为关系较为紧密的团体，这就导致了原理层面上的矛盾。

不过，当我们对数据进行了 OneHot Encoding 后，由于所有的特征划分出来的子数据集都最多只能有两个（因为所有特征都变成了二值特征），所以 scikit-learn 的这种做法也就无伤大雅了。

2.2.7* 决策树与回归

将决策树从分类问题过渡到回归问题其实可能会比想象中的简单。事实上不知大家是否发现，我们在整个 2.2 节中没有出现“类别”这两个字；取而代之的，我们一直用的是“标签”这两个字。所以想用决策树做回归问题的话，只需把相关的关键点进行更改即可，整体的大思路——不断地划分特征向量空间，直至每个子空间都足以做出决策并被贴上相应的标签——是没有变的。

首先要更改的是 \tilde{H} 函数的定义。 \tilde{H} 衡量的是数据集的信息量，注意我们曾在 2.2.2 节的开始说过，信息量是“不确定性”的相反数。对于回归问题而言，很自然的想法就是把当前最好的标签和数据集中的标签的“距离”作为不确定性的度量。具体而言，假设我们当前的数据集为

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$$

那么我们就要找到

$$\hat{y} = \arg \min_y \sum_{i=1}^N \|y_i - y\|^2$$

并把 \tilde{H} 定义成

$$\tilde{H}(D) \triangleq \sum_{i=1}^N \|y_i - \hat{y}\|^2$$

通过简单的求导和解方程，不难得出

$$\hat{y} = \frac{1}{N} \sum_{i=1}^N y_i$$

即 \hat{y} 应该是 D 中标签的均值。可以证明，此时 \hat{y} 其实就是 D 中标签的期望的极大似然估计，从而 \tilde{H} 其实就是 D 中标签的方差（Variance）的极大似然估计。

在有了 \tilde{H} 后，定义 $\Delta\tilde{H}$ 就很“顺其自然”了。注意到

$$\Delta\tilde{H}(D; x^{(i)}) \triangleq w_0(D)\tilde{H}(D) - \sum_{j=1}^q w_j(D_j)\tilde{H}(D_j)$$

我们可以简单地将权重取得和 ID3 算法中的一致，即

$$w_0(D) \triangleq 1, \quad w_j(D_j) \triangleq \frac{|D_j|}{|D|}$$

从而就有

$$\begin{aligned} \Delta\tilde{H}(D; x^{(i)}) &\triangleq \tilde{H}(D) - \sum_{j=1}^q \frac{|D_j|}{|D|} \tilde{H}(D_j) \\ &= \sum_{i=1}^N \left\| y_i - \frac{1}{N} \sum_{j=1}^N y_j \right\|^2 - \sum_{i=1}^q \frac{|D_i|}{|D|} \sum_{j=1}^{|D_i|} \left\| y_{ij} - \frac{1}{|D_i|} \sum_{k=1}^{|D_i|} y_{ik} \right\|^2 \end{aligned}$$

最后需要注意的是，在分类问题中，当我们认为一个子空间足以做出决策时，我们会将对应的子数据集中出现最多次的标签作为该子空间的标签。而在回归问题中不能这样做，取而代之的是用当前子数据集中标签的均值来作为该子空间的标签。这样做的合理性和必要性在上面推导 \tilde{H} 的过程中就已经体现出来了，这里不再赘述。

综上所述，回归问题和分类问题相比，只有如下两点不同：

- 分类问题的 \tilde{H} 是 Gini Index 或 Entropy，回归问题的则是 Variance。
- 分类问题在打标签时会选用当前子数据集中出现次数最多的标签，回归问题则会选用当前子数据集所有标签的平均值。

以上，我们就对决策树算法的抽象逻辑与实际应用都做了简要的说明。

2.3 支持向量机

对于 2.1 节讲解的朴素贝叶斯算法，核心公式是

$$p(y_k) = \frac{N_k}{N}, \quad k = 1, 2, \dots, K$$

$$p(x^{(i)}|y_k; \theta) = \frac{N_k^{(i)}}{N_k}, \quad i = 1, 2, \dots, n; k = 1, 2, \dots, K$$

对于 2.1 节讲解的决策树算法，核心公式是（以 ID3 算法为例）

$$\Delta \tilde{H}^{ID3}(D; x^{(i)}) = - \sum_{i=1}^K \frac{N_i}{N} \log \frac{N_i}{N} + \sum_{j=1}^q \frac{|D_j|}{|D|} \sum_{i=1}^K \frac{N_{ji}}{N_j} \log \frac{N_{ji}}{N_j}$$

可以看到，它们的本质都只是对输入数据集做了一些“计数”的操作。但不难想象的是，机器学习肯定不只是“计数”这么简单的事。为此，我们会在本章介绍另一类非常重要的训练方法——梯度下降（Gradient Descent）。梯度下降在神经网络中的地位（到目前为止来看）是非常高的，毫不夸张地说，如果没有梯度下降，神经网络可能就无法走到今天这个地步。

由简入繁，我们先通过两个能够简单应用梯度下降的模型——感知机（Perceptron）和支持向量机（Support Vector Machine，常简称为 SVM）来大致感受一下梯度下降的应用方式。具体而言，我们会先在 2.3.1 节简要介绍感知机和 SVM 的大致思想，然后会在 2.3.2 节给出它们算法的原始形式，继而会在 2.3.3 节介绍梯度下降的框架以及该框架下它们的具体训练算法。我们会在 2.3.4 节补充说明如何将它们转化为非线性模型，并在最后的 2.3.5 节给出具体的应用方法。

2.3.1 分离超平面与几何间隔

为了更好地介绍感知机和支持向量机（SVM）的思想与模型细节，我们需要先介绍一下“超平面（Hyperplane）”这个概念。在我们比较熟悉的二维平面和三维空间中，超平面的概念是很好理解的：

- 二维平面的超平面即为一维直线。
- 三维空间的超平面即为二维平面。

那么拓展到高维空间时，也可以类似地进行定义：在一个 n 维的空间中，我们可以把超平面定义成 $n-1$ 维的“某个东西”。

注意到直线方程是

$$w_0 + w_1 x^{(1)} + w_2 x^{(2)} = 0$$

平面方程是

$$w_0 + w_1x^{(1)} + w_2x^{(2)} + w_3x^{(3)} = 0$$

所以很自然的想法就是， $n-1$ 维的“某个东西”应该定义成

$$w_0 + \sum_{i=1}^n w_i x^{(i)} = 0$$

事实上，这确实就是超平面的数学定义。而我们之所以将上面这个方程称为“超平面”，是因为它高维空间中的表现从几何的角度来看确实很像一个平面，比如说它本身是线性的，而且能将高维空间分成互不相交的两个子空间。

我们通常会用字母 Π 来代指超平面，并把相应的超平面方程简记为 $f_{\Pi}(\mathbf{x}) = 0$ ，即

$$\Pi: f_{\Pi}(\mathbf{x}) = 0$$

其中

$$\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^T$$

是我们的特征向量。

$$f_{\Pi}(\mathbf{x}) \triangleq w_0 + \sum_{i=1}^n w_i x^{(i)}$$

则是我们的超平面公式。在此基础上，如果令

$$\mathbf{w} \triangleq (w_1, w_2, \dots, w_n)^T$$

那么超平面公式就能写成

$$f_{\Pi}(\mathbf{x}) = w_0 + \mathbf{w}^T \mathbf{x}$$

利用超平面公式，我们可以很自然地将被超平面划分出来的两个子空间分别表述为如下的形式（假设原始空间为 \mathcal{X} ，两个子空间分别为 \mathcal{X}_1 和 \mathcal{X}_2 ）：

$$\mathcal{X}_1 = \{\mathbf{x} | \mathbf{x} \in \mathcal{X} \wedge f_{\Pi}(\mathbf{x}) < 0\}$$

$$\mathcal{X}_2 = \{\mathbf{x} | \mathbf{x} \in \mathcal{X} \wedge f_{\Pi}(\mathbf{x}) \geq 0\}$$

我们可以通过二维平面上的例子来直观理解超平面的这些概念（如图 2.3 和图 2.4 所示）。在图 2.3 与图 2.4 中， $y = x + 1$ 、 $x = 0.5$ 就分别是所在二维平面的超平面，它们的左边和右边则分别对应着 $f_{\Pi}(\mathbf{x}) < 0$ 、 $f_{\Pi}(\mathbf{x}) \geq 0$ 的区域：

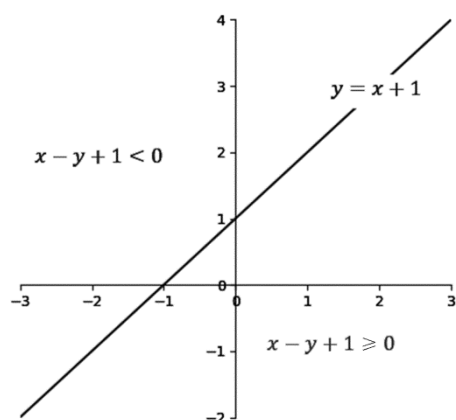


图2.3 超平面示意图 (1)

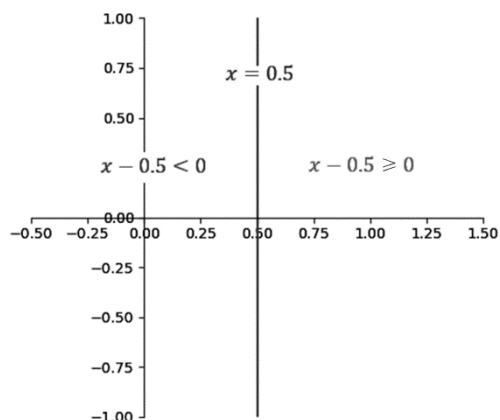


图2.4 超平面示意图 (2)

需要指出的是，并非所有超平面的左边都对应着 $f_H(\mathbf{x}) < 0$ 的区域。仍以二维平面为例，当直线表达式中 \mathbf{x} 对应的系数小于0时，事实上就是超平面的右边才对应着 $f_H(\mathbf{x}) < 0$ 的区域，如图2.5所示。

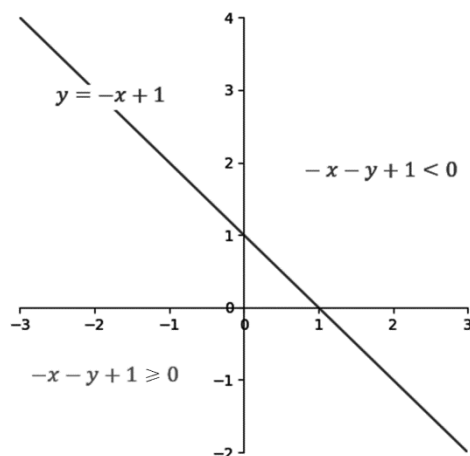


图2.5 超平面示意图 (3)

注意：当空间维度变高时，我们其实很难直观地定义出“左”和“右”的概念。之所以在图2.3到图2.5的例子中用了“左”和“右”的概念，只是为了帮助大家直观地理解超平面与其划分出来的两个子空间 \mathcal{X}_1 和 \mathcal{X}_2 之间的关系而已。

在理解了超平面的一些基本内涵后，我们就能开始讨论感知机和SVM的一些基本性质了。首先无论是感知机模型还是SVM，它们原始针对的任务都是连续型特征向量（即所有特征都是连续型特征）下的二分类问题。换句话说，它们所针对的数据集都是二分类连续型数据集：

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$$

其中对任意的特征向量 \mathbf{x}_j ，它的 n 个特征 $x_j^{(1)}, x_j^{(2)}, \dots, x_j^{(n)}$ 都是连续型特征；且对任意的标

签 y_j ，它可能的取值都只有两个，即

$$y_j \in \mathcal{Y} = \{c_1, c_2\}$$

然后，感知机和 SVM 的核心思想则都是：在特征向量空间中找到一个超平面，使得它能将两个类别的样本划分开。由于该超平面的作用是将两个类别的样本分离，所以我们一般会称它为“分离超平面”，于是感知机和 SVM 的核心思想即在于找到一个合适的分离超平面。从这个角度来看，它们和决策树做的事情类似，也是在划分特征向量空间；不过和决策树相比，它们的做法有两大不同之处：

- 决策树一般会将特征向量空间划分成很多个子空间，感知机和 SVM 则只会划分出两个子空间。
- 决策树（在所有特征都是连续型特征时）在划分空间时，用于划分的超平面都是垂直于某个维度的，而感知机和 SVM 的分离超平面则可以是任意一个超平面。

之所以说决策树的“超平面垂直某个维度”，是因为它的划分方法为

$$D \xrightarrow{x^{(i)}} \{D_1, D_2\}$$

其中

$$D_1 = \{(\mathbf{x}, y) \mid (\mathbf{x}, y) \in D \wedge x^{(i)} < \tilde{\epsilon}^{(i)}\}$$

$$D_2 = \{(\mathbf{x}, y) \mid (\mathbf{x}, y) \in D \wedge x^{(i)} \geq \tilde{\epsilon}^{(i)}\}$$

注意， $x^{(i)} = \tilde{\epsilon}^{(i)}$ 其实是一个超平面，因为只需令

$$w_0 \triangleq -\tilde{\epsilon}^{(i)}, \quad w_i \triangleq 1, \quad w_j \triangleq 0 \ (j \neq 0 \wedge j \neq i)$$

那么超平面方程

$$f_{\Pi}(\mathbf{x}) = w_0 + \sum_{j=1}^n w_j x^{(j)} = 0$$

就能转化为

$$x^{(i)} - \tilde{\epsilon}^{(i)} = 0$$

即

$$x^{(i)} = \tilde{\epsilon}^{(i)}$$

这就说明 D_1 、 D_2 确实可以看作被超平面 $x^{(i)} = \tilde{\epsilon}^{(i)}$ 划分出来的两个子空间，而超平面 $x^{(i)} = \tilde{\epsilon}^{(i)}$ 显然是垂直于第 i 个维度的（可以配合图2.4来理解）。

注意：使得划分超平面不垂直于某个维度的决策树算法是存在的，不过对于本书所介绍的决策树算法而言，相应的划分超平面都是垂直于某个维度的。

而对感知机和 SVM 而言，既然它们只进行一次划分，它们的（分离）超平面方程自然要比决策树的自由许多。事实上，它们会把 w_0, w_1, \dots, w_n 都视为可训练的参数，而不像决策树那样，每次划分时会先把某个 w_i 设为 1、其余 w_j 设为 0，然后只把 w_0 视为可训练的参数。

在训练完 w_0, w_1, \dots, w_n 这些参数之后，由前文的讨论，我们就能自然地把模型 G 给定义出来了。具体而言，假设我们现在的任务是一个二分类问题，两个类别分别是 c_1 、 c_2 ，那么 G 的表达式就能直观地写成

$$G(x) \triangleq \begin{cases} c_1 & , \text{ if } f_{\Pi}(x) = w_0 + \mathbf{w}^T \mathbf{x} < 0 \\ c_2 & , \text{ if } f_{\Pi}(x) = w_0 + \mathbf{w}^T \mathbf{x} \geq 0 \end{cases}, \quad \forall \mathbf{x} \in \mathcal{X}$$

其中 \mathbf{w} 的定义和前文所叙述的一致，即

$$\mathbf{w} \triangleq (w_1, \dots, w_n)^T$$

不难看出， G 在特征向量 \mathbf{x} 上输出的类别完全取决于分离超平面 Π ，所以只要我们完成了对参数 w_0 和 \mathbf{w} 的训练（或说，只要它们足够好），就能够定出分离超平面 Π ，从而就能根据上面这个公式来完成模型的搭建了。为了便于讨论和简化公式的书写，我们一般会令

$$c_1 \triangleq -1, \quad c_2 \triangleq 1, \quad \tilde{\mathbf{w}} \triangleq \{w_0, \mathbf{w}\}$$

从而 G 的参数就可以统一为 $\tilde{\mathbf{w}}$ ，且它的表达式就能写成

$$G(x) \triangleq \begin{cases} -1 & , \text{ if } f_{\Pi}(x) < 0 \\ 1 & , \text{ if } f_{\Pi}(x) \geq 0 \end{cases} = \text{sign}(f_{\Pi}(x)), \quad \forall \mathbf{x} \in \mathcal{X}$$

其中， sign 是符号函数，当输入小于 0 时输出 -1，否则输出 1：

$$\text{sign}(x) = \begin{cases} -1 & , \text{ if } x < 0 \\ 1 & , \text{ if } x \geq 0 \end{cases}$$

从而对于给定的参数 $\tilde{\mathbf{w}}$ ， G 的表达式即为

$$G(x) = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$$

可以看到它非常简洁。所以在本节之后的讨论中，如果没有特别指出的话，就统一认为二分类数据集中所有标签的取值都为 -1 或 1。

剩下的关键问题在于，究竟怎样的参数 $\tilde{\mathbf{w}}$ 才算是足够好的呢？对于感知机来说，它认为对于给定的二分类连续型数据集 D ，只要能找到 $\tilde{\mathbf{w}}$ ，使得对应的模型 G 满足条件

$$G(\mathbf{x}_i) = y_i, \quad \forall (\mathbf{x}_i, y_i) \in D$$

那么该 $\tilde{\mathbf{w}}$ 就是足够好的。这其实非常直观：我们的 G 既不复杂（只将特征向量空间划分成了两份），又能在训练数据集 D 上有完美的表现，那它当然是足够好的。而且可以证明的是，如果一个数据集 D 从理论上存在着某个参数 $\tilde{\mathbf{w}}$ ，使得对应的模型 G 满足上述条件的话，那么就会有一套算法，保证我们可以在有限的时间内找到某个足够好的参数 $\tilde{\mathbf{w}}^*$ （ $\tilde{\mathbf{w}}^*$ 和 $\tilde{\mathbf{w}}$ 不一定相等），它能使对应的模型 G 满足上述条件，即此时 G 能够对 D 进行完美分类。像这种能被感知机完美分类的数据集 D ，我们通常称它为“线性可分的数据集”。

不过遗憾的是，虽然对一个线性可分的数据集，感知机确实能够找到一个可以进行完美分类的参数 $\tilde{\mathbf{w}}$ ，但它找出来的这个参数一般而言都比较差。换句话说，它对自己的要求有些“太弱”了。为了更深刻地理解这里面的逻辑，我们需要引入“间隔（Margin）”的概念。

所谓间隔，我们可以直观地理解为“点到超平面的（带符号的）距离”。回忆在本小节开头介绍超平面时我们曾经说过，超平面方程对应着一张超平面：

$$\Pi: f_{\Pi}(\mathbf{x}) = 0$$

那么一个很自然的疑问就是：当超平面公式 $f_{\Pi}(\mathbf{x})$ 不等于0时，它背后的意义究竟是什么？事实上对于二分类样本空间 \mathcal{D} 中的某个样本 (\mathbf{x}, y) 和某个超平面而言， $f_{\Pi}(\mathbf{x})$ 的函数值与 y 的乘积正是 \mathbf{x} 到 Π 的“函数间隔（Functional Margin，可以简称为FM）”，即

$$\text{FM}(\mathbf{x}, \Pi) \triangleq y \cdot f_{\Pi}(\mathbf{x}) = y \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x})$$

之所以要乘上一个 y （注意， y 只有-1或1这两种取值），是因为我们希望函数间隔能够反映出 (\mathbf{x}, y) 在 Π 作为分离超平面时的表现。具体而言，对于我们现在定义的FM来说，不难看出它具有以下两个性质：

- 当 $G(\mathbf{x}) = \text{sign}(f_{\Pi}(\mathbf{x})) = y$ 时， $f_{\Pi}(\mathbf{x}) = \mathbf{w}_0 + \mathbf{w}^T \mathbf{x}$ 和 y 同号，从而 $\text{FM}(\mathbf{x}, \Pi) > 0$ 。
- 当 $G(\mathbf{x}) \neq y$ 时，同理可推出 $\text{FM}(\mathbf{x}, \Pi) < 0$ 。

换句话说，就是当模型做出正确、错误的分类时，函数间隔FM分别是正数、负数。也正是因为间隔可以是负数，所以我们前面才说间隔是“带符号的距离”。

虽说从事后诸葛亮的角度来看，这种定义方式确实挺有道理，不过它到底是怎样被得出来的呢？下面就来分析一下背后的逻辑。事实上我们在定义样本 (\mathbf{x}, y) 到平面 Π 的间隔 d 时，如果我们想让 d 拥有“反映出 (\mathbf{x}, y) 在 Π 作为分离超平面时的表现”的功能的话，从直观上来说都会这样做，示意图如图2.6所示：

- 将 \mathbf{x} （垂直）投影到 Π 上
- 设投影点为 \mathbf{x}^* ，则将间隔定义为

$$d((\mathbf{x}, y), \Pi) \triangleq \begin{cases} \|\mathbf{x} - \mathbf{x}^*\|^2 & , \text{ if } G(\mathbf{x}) = y \\ -\|\mathbf{x} - \mathbf{x}^*\|^2 & , \text{ if } G(\mathbf{x}) \neq y \end{cases}$$

即我们前面所说的，将间隔定义为“带符号的距离”。

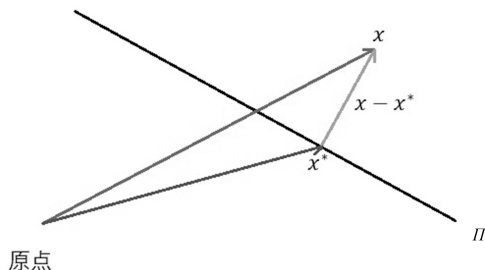


图2.6 间隔的定义方式（1）

那么为了找到垂直投影，我们得先找到垂直于超平面 Π 的方向，不难看出 \mathbf{w} 其实就是垂直于 Π 的，这是因为对 $\forall \mathbf{x}_1, \mathbf{x}_2 \in \Pi$ ，由

$$\begin{cases} \mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_1 = 0 \\ \mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_2 = 0 \end{cases}$$

可知（两式相减即可）

$$\mathbf{w}^T (\mathbf{x}_1 - \mathbf{x}_2) = 0$$

即 \mathbf{w} 垂直于 $\mathbf{x}_1 - \mathbf{x}_2$ 这个方向，从而也就垂直于 Π ，示意图如图 2.7 所示。

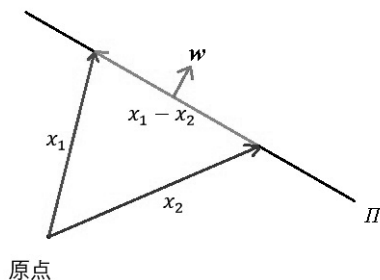


图2.7 间隔的定义方式（2）

那么结合图 2.6，我们可以设

$$\mathbf{x} - \mathbf{x}^* = \lambda \mathbf{w}$$

其中 λ 可正可负，于是就有（注意，由 $\mathbf{x}^* \in \Pi$ 可知 $\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}^* = 0$ ）

$$\begin{aligned} \|\mathbf{x} - \mathbf{x}^*\|^2 &= (\mathbf{x} - \mathbf{x}^*)^T (\mathbf{x} - \mathbf{x}^*) = \lambda \mathbf{w}^T (\mathbf{x} - \mathbf{x}^*) \\ &= \lambda [\mathbf{w}^T (\mathbf{x} - \mathbf{x}^*) + (\mathbf{w}_0 - \mathbf{w}_0)] \\ &= \lambda [\mathbf{w}_0 + \mathbf{w}^T \mathbf{x} - (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}^*)] \\ &= \lambda (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}) \\ &= \lambda \cdot f_{\Pi}(\mathbf{x}) \end{aligned}$$

从而

$$d((\mathbf{x}, y), \Pi) = \begin{cases} \lambda \cdot f_{\Pi}(\mathbf{x}) & , \text{ if } G(\mathbf{x}) = y \\ -\lambda \cdot f_{\Pi}(\mathbf{x}) & , \text{ if } G(\mathbf{x}) \neq y \end{cases}$$

注意到

$$G(\mathbf{x}) = \text{sign}(f_{\Pi}(\mathbf{x}))$$

且由 $\|\mathbf{x} - \mathbf{x}^*\|^2 = \lambda \cdot f_{\Pi}(\mathbf{x})$ 和 $\|\mathbf{x} - \mathbf{x}^*\|^2 \geq 0$ 知， λ 的正负性其实和 $f_{\Pi}(\mathbf{x})$ 的正负性一致，于是我们就能把 d 简化为

$$d((\mathbf{x}, y), \Pi) = \lambda y \cdot f_{\Pi}(\mathbf{x})$$

可以看到，当取 $\lambda = 1$ 时，就有 $d((\mathbf{x}, y), \Pi) = \text{FM}(\mathbf{x}, \Pi)$ 。

虽然上述推导看上去挺优雅的，但是其实里面隐藏着一个大问题：试想我们把参数 $\tilde{\mathbf{w}}$ 增大 k 倍（即 \mathbf{w}_0 和 \mathbf{w} 同时增大 k 倍），那么新得到的超平面 $\tilde{\Pi}: k\mathbf{w}_0 + (k\mathbf{w})^T \mathbf{x} = 0$ 其实是等价于原超平面 Π 的：

$$\mathbf{x} \in \tilde{\Pi} \Leftrightarrow k\mathbf{w}_0 + (k\mathbf{w})^T \mathbf{x} = 0 \Leftrightarrow \mathbf{w}_0 + \mathbf{w}^T \mathbf{x} = 0 \Leftrightarrow \mathbf{x} \in \Pi$$

但此时的 $d((\mathbf{x}, y), \Pi)$ 却会直接增大（或减小） k 倍：

$$\begin{aligned} d((\mathbf{x}, y), \tilde{\Pi}) &= \lambda y \cdot f_{\tilde{\Pi}}(\mathbf{x}) \\ &= \lambda y \cdot [k\mathbf{w}_0 + (k\mathbf{w})^T \mathbf{x}] \\ &= k\lambda y \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}) \\ &= k \cdot d((\mathbf{x}, y), \Pi) \end{aligned}$$

极端的情况是，当 $\tilde{\mathbf{w}}$ 增大无穷倍时，明明超平面没变，间隔却跟着增大了无穷倍，这当然是不合理的。所以我们需要尝试把这种缩放的影响给抹去，常见的做法就是做某种意义上的归一化：

$$\tilde{d}((\mathbf{x}, y), \Pi) = \frac{1}{\|\mathbf{w}\|} y \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x})$$

注意：由于缩放的影响已经被抹去，所以有着缩放系数意义的 λ 也就跟着被抹去了。

这样定义的 \tilde{d} 通常被称为“几何间隔（Geometric Margin，可以简称为 GM）”，在使用间隔这个概念时，我们一般用的都是它。

在知道了间隔的定义后，我们就能比较严谨地说明为何感知机的要求对于线性可分的数据集而言又“太弱”了。事实上，即使是一个非常简单的二维数据集，感知机也可能会表现得非常不合理，如图 2.8 所示。

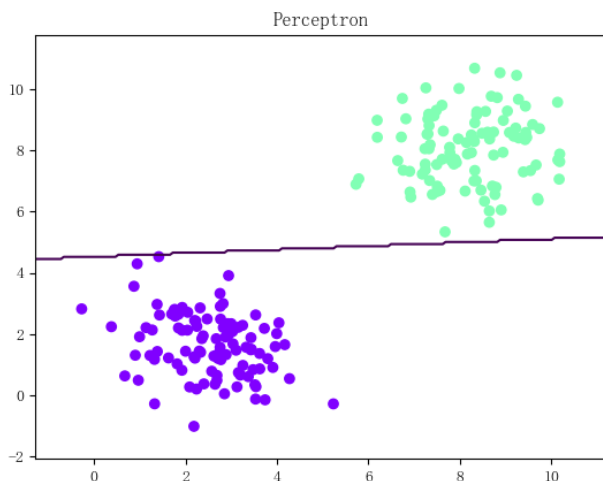


图2.8 感知机在二维线性可分数据集上的不良表现

之所以我们从直观上认为图 2.8 所示的分离超平面（即中间那条黑线）不是一个很好的分离超平面，是因为它离两类样本点都“太近”了。换句话说，两类样本点离分离超平面的几何间隔太小，导致该分离超平面的泛化能力有可能很差。而感知机只要求对样本进行正确分类，没有对几何间隔做出任何要求，这无疑是不合理的，也是我们说它太弱的原因。

那么解决这种问题的思路也就清晰了：只需在训练过程中把几何间隔也纳入考虑范围即可。而这其实就是 SVM 与感知机的唯一不同之处，也是 SVM 的核心思想所在。在该思想下，上述二维线性可分数据集就能被很好地解决，如图 2.9 所示。

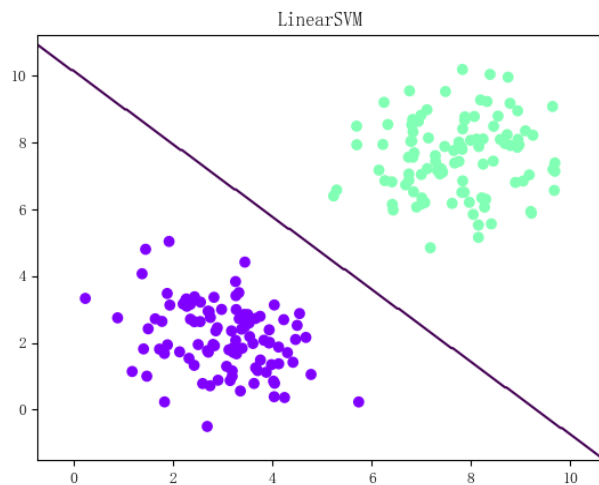


图2.9 SVM在二维线性可分数据集上的优良表现

2.3.2* 感知机与 SVM 的原始形式

在 2.3.1 节中，我们用了比较直观的说法来对感知机和 SVM 做了介绍。在本节中，我们则会较为系统地介绍感知机和 SVM 算法的原始形式。由于本节公式较多较难，且大部分内容和本书主题并不直接相关，所以即使无法全部理解也没有关系，只需对本节最后给出的结论有一个印象即可。

我们在第 1 章里面曾经说过，在机器学习领域中，我们通常会在训练集 D 中定义一个全局的代价函数 C ，它是损失函数 L 在 D 上的均值：

$$C(G, D) = \frac{1}{N} \sum_{i=1}^N L(G(\mathbf{x}_i), y_i)$$

但是在前两节中，我们却似乎并没有用到损失函数和代价函数（Loss and Cost）。这是因为前两节所用的训练方法从本质上来说都是计数（正如本节开头所说的那样），所以不显式定义出 Loss and Cost 也无伤大雅。而如果我们想要应用梯度下降法的话，Loss and Cost 就是不可或缺的组件了。

注意：朴素贝叶斯和决策树也是有损失函数的。具体而言，算法 2.1 所对应的（原始形式的）朴素贝叶斯的损失函数是比较简单的 0-1 损失函数：

$$C(G, D) = \frac{1}{N} \sum_{i=1}^N L(G(\mathbf{x}_i), y_i) = \frac{1}{N} \sum_{i=1}^N I(G(\mathbf{x}_i) \neq y_i)$$

决策树的代价函数则是数据集的信息量（分类问题）或方差（回归问题）。至于为什么朴素贝叶斯的损失函数是 0-1 损失函数，其实是因为我们可以证明，应用朴素贝叶斯算法得到的模型所做的决策，就是 0-1 损失函数下的贝叶斯决策。有兴趣的读者可以阅读 <http://www.carefree0910.com/posts/e312d61a/> 及一些相关的文章，这里就不展开叙述了。

所以问题的关键所在就成了如何把 Loss and Cost 定义出来。先看感知机，由 2.3 节的介绍我们知道，感知机只关心“把所有样本分对”而不关心“把所有样本分好”。再联系到几何间隔的性质，我们可以自然地把感知机的 Loss 定义为

$$L_P(G(\mathbf{x}_i), y_i) = \begin{cases} 0 & , \text{ if } G(\mathbf{x}_i) = y_i \\ |\tilde{d}((\mathbf{x}_i, y_i), \Pi)| & , \text{ if } G(\mathbf{x}_i) \neq y_i \end{cases}$$

换句话说，对于每个样本点 (\mathbf{x}_i, y_i) ，当模型正确地进行了分类时，就认为损失是 0，否则，就认为 (\mathbf{x}_i, y_i) 与模型对应的分离超平面 Π 之间的几何间隔的绝对值越大（即 (\mathbf{x}_i, y_i) 离 Π 越远）、损失就越大，这是符合直观的。

注意，函数间隔 FM 拥有“当模型做出正确、错误的分类时，FM 分别是正数、负数”这个性质，而几何间隔 GM 是 FM 的 $\frac{1}{\|\mathbf{w}\|}$ 倍，所以也拥有这个性质，从而上述损失函数就能改写成

$$L_P(G(\mathbf{x}_i), y_i) = |-\tilde{d}((\mathbf{x}_i, y_i), \Pi)|_+$$

其中， $|\cdot|_+$ 是函数 $\max(\cdot, 0)$ 的简写：

$$|x|_+ = \begin{cases} x & , \quad x \geq 0 \\ 0 & , \quad x < 0 \end{cases}$$

注意到

$$\tilde{d}((\mathbf{x}_i, y_i), \Pi) = \frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)$$

从而

$$L_P(G(\mathbf{x}_i), y_i) = \frac{1}{\|\mathbf{w}\|} |-y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

在有了 Loss 后，代价函数就是简单地求一个均值而已：

$$C_P(G, D) = \frac{1}{N} \sum_{i=1}^N L_P(G(\mathbf{x}_i), y_i) = \frac{1}{N \|\mathbf{w}\|} \sum_{i=1}^N |-y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

以上就是感知机的 Loss and Cost 的形式。

对于 SVM 而言，我们在 2.3.1 节的最后曾经说过，它和感知机的唯一区别就在于它要在训练过程中，把样本点 (\mathbf{x}_i, y_i) 到分离超平面 Π 的几何间隔也纳入考虑范围。表现在损失函数里面的话，其实可以直观地“在损失函数里加入对几何间隔太小的惩罚”。具体而言，我们可以认为，只有在模型正确地进行了分类且相应的几何间隔大于某个阈值 M 时，损失才是 0；否则，我们都要对模型做出一些惩罚：

$$L_{SVM}(G(\mathbf{x}_i), y_i) = \begin{cases} 0 & , \quad \text{if } G(\mathbf{x}_i) = y_i \wedge |\tilde{d}((\mathbf{x}_i, y_i), \Pi)| \geq M \\ M - |\tilde{d}((\mathbf{x}_i, y_i), \Pi)| & , \quad \text{if } G(\mathbf{x}_i) = y_i \wedge |\tilde{d}((\mathbf{x}_i, y_i), \Pi)| < M \\ M + |\tilde{d}((\mathbf{x}_i, y_i), \Pi)| & , \quad \text{if } G(\mathbf{x}_i) \neq y_i \end{cases}$$

类似的，上式可以简写成

$$L_{SVM}(G(\mathbf{x}_i), y_i) = \left| M - \frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) \right|_+$$

下面的问题就在于如何定出 M 了。为了解决该问题，我们可以换一个角度来叙述 SVM 的思想。我们可以把 SVM 强调“几何间隔要足够大”的这个思想，转化为“最大化最小的几何间隔”这个优化问题。换句话说，假设现在的训练集为 D ，那么 SVM 的目标就可以概括为如下两点：

- 让所有样本都被正确分类，即

$$y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) > 0, \quad \forall (\mathbf{x}_i, y_i) \in D$$

- 让样本点到分离超平面的最小几何间隔尽可能大，即

$$\max \min_{(\mathbf{x}_i, y_i) \in D} \frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)$$

注意， $y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) > 0$ 的性质和 $\frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)$ 的值在 w_0 和 \mathbf{w} 同时扩大 k 倍时不会改变，所以我们完全可以假设：

- 若 $(\mathbf{x}_p, y_p) = \arg \min_{(\mathbf{x}_i, y_i) \in D} \frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)$ ，则

$$y_p \cdot (w_0 + \mathbf{w}^T \mathbf{x}_p) = 1$$

否则，假设

$$y_p \cdot (w_0 + \mathbf{w}^T \mathbf{x}_p) = c$$

那么令

$$w_0 \leftarrow \frac{w_0}{c}, \quad \mathbf{w} \leftarrow \frac{\mathbf{w}}{c}$$

即可。在此假设下，显然就有

$$\max \min_{(\mathbf{x}_i, y_i) \in D} \frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) = \max \frac{1}{\|\mathbf{w}\|} y_p \cdot (w_0 + \mathbf{w}^T \mathbf{x}_p) = \max \frac{1}{\|\mathbf{w}\|}$$

同时，注意由于

$$(\mathbf{x}_p, y_p) = \arg \min_{(\mathbf{x}_i, y_i) \in D} \frac{1}{\|\mathbf{w}\|} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)$$

这个最小化过程中 \mathbf{w} 是固定的，所以可以把 $\frac{1}{\|\mathbf{w}\|}$ 这一项去掉，从而有

$$(\mathbf{x}_p, y_p) = \arg \min_{(\mathbf{x}_i, y_i) \in D} y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)$$

所以就有

$$y_p \cdot (w_0 + \mathbf{w}^T \mathbf{x}_p) = 1 \Rightarrow y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) \geq 1, \forall (\mathbf{x}_i, y_i) \in D$$

于是 SVM 的目标就转化为这样一个（带约束的）优化问题：

$$\max_{w_0, \mathbf{w}} \frac{1}{\|\mathbf{w}\|}$$

$$s.t. y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) \geq 1, \forall (\mathbf{x}_i, y_i) \in D$$

亦即

$$\min_{w_0, \mathbf{w}} \frac{\|\mathbf{w}\|^2}{2}$$

$$s.t. y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) \geq 1, \quad \forall (\mathbf{x}_i, y_i) \in D$$

不过该优化问题的约束稍显苛刻（换句话说，就是 $y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) \geq 1$ 这个约束太“硬”了）。事实上当数据集线性不可分时，该优化问题是必定无解的，这就会导致模型震荡。所以为了让 SVM 在线性不可分的数据上仍有不错的表现，从直观上来说，我们应该“放松”对模型的限制，或说让模型的约束“软”一点：

$$\min_{w_0, \mathbf{w}} \frac{\|\mathbf{w}\|^2}{2}$$

$$s.t. y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i) \geq 1 - \xi_i, \quad \forall (\mathbf{x}_i, y_i) \in D$$

其中

$$\xi_i \geq 0, \quad \forall i = 1, 2, \dots, N$$

当然，仅仅放松限制很有可能会使模型倾向于选择将限制放松得越来越厉害（比如令

$\xi_i \rightarrow \infty$), 所以我們还需要让这种放松受到惩罚。具体而言, 可以把优化问题改为:

$$\min_{w_0, \mathbf{w}} \left[\frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N \xi_i \right]$$

$$s. t. y_i \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_i) \geq 1 - \xi_i, \quad \forall (\mathbf{x}_i, y_i) \in D$$

其中 C 是一个常数, 可以把它理解为对放松的“惩罚力度”。可以证明, 上述带约束的优化问题在经过数学变换后, 等价于下面这个无约束的优化问题:

$$\min_{w_0, \mathbf{w}} \left[\frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N |1 - y_i \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_i)|_+ \right]$$

所以我們就可以很自然地把 SVM 的代价函数定义为

$$C_{SVM}(G, D) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N |1 - y_i \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

以上就是 SVM 的 Loss and Cost 的形式, 于是至此我们就完成了感知机与 SVM 的损失函数和代价函数的推导。

将前面的所有讨论进行一个汇总, 不难得出如下几个关键点:

- 感知机和 SVM 都是线性模型, 它们都对对应着一张分离超平面 Π :

$$\Pi: \mathbf{w}_0 + \mathbf{w}^T \mathbf{x} = 0$$

- 感知机和 SVM 模型的表达式都是:

$$G(\mathbf{x}) = \text{sign}(\mathbf{w}_0 + \mathbf{w}^T \mathbf{x})$$

- 感知机和 SVM 的代价函数分别是:

$$\begin{cases} C_P(G, D) = \frac{1}{N \|\mathbf{w}\|} \sum_{i=1}^N | -y_i \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_i) |_+ \\ C_{SVM}(G, D) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N |1 - y_i \cdot (\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}_i)|_+ \end{cases}$$

从而感知机和 SVM 的训练过程, 就可以看作求解最小化这两个代价函数所对应的优化问题的过程。我們通常会称这两个优化问题为感知机和 SVM 的原始形式。

2.3.3 梯度下降法

在知道了模型的原始形式后, 我們就要寻求一种能够加以训练的算法了。对于感知机和 SVM (以及本书的主题——神经网络) 这种能将问题转化为无约束的最小化损失函数的模型而言, 梯度下降法常常是一种上手快、效率高的算法。这一方面得益于其简单的形式, 另一方面

则得益于现有的深度学习框架（比如 TensorFlow）基本都含有自动求导并更新参数的功能，所以我们往往不用自己操心梯度下降法的种种实现技巧，而可以将注意力集中在模型的构造和搭建上。

接下来叙述一下梯度下降法的大体框架。为了理解梯度下降法，我们首先需要明确的当然就是“梯度”这两个字究竟是什么意思。从直观上来说，我们只需知道这一点：梯度是使得函数值上升最快的方向。

这很容易让我们联想到函数的导数。事实上如果不追求严谨的话，梯度确实就是“函数对高维张量的导数”。如果大家想了解梯度的严谨定义而不满足于上述的直观解释的话，可以参见维基百科中的详细定义：<https://en.wikipedia.org/wiki/Gradient>。

注意，我们的目的是最小化损失函数，所以一个很自然的想法就是：如果梯度是使得函数值上升最快的方向，那么梯度的相反数（即负梯度）是否就是使得函数值下降最快的方向呢？答案是肯定的，而这也是为何有时我们会看到梯度下降法被称作“最速下降法（Steepest Descent）”的原因——它会在每一步中计算出当前损失函数的梯度，并沿着负梯度的方向向前迈进一步，以期能够使损失函数下降得最快。由此不难看出，梯度下降法的核心问题就归结为如下两点：

- 如何进行梯度的计算？
- “向前迈进一步”中，步长到底应该多大？

如果用机器学习的术语来说的话，上面两点可以更本质地叙述为如下两点：

- 如何获得参数更新的方向？
- 学习速率（Learning Rate）应该如何设置？

下面我们就来进行相应的讨论。我们会要求大家拥有简单的函数求导的能力，但不会做出除此之外更多的要求。为了方便叙述，在此先做一些符号规定：

- 我们的目标是最小化损失函数 L 。
- L 受参数 $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_n$ 的影响，它们或是向量或是矩阵，但不会是更高维的张量。
- 统一使用 “ $\nabla_{\mathbf{w}_i} L$ ” 表示 “ \mathbf{w}_i 在当前 L 下的梯度”。

由于我们对普通函数的求导比较熟悉，所以一切推导的出发点都会是普通函数的求导。由于我们假设了所有参数 $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_n$ 都是向量或矩阵，所以我们就先从向量和矩阵的直接求梯度入手。

先来看向量的求梯度方法，假设参数 \mathbf{w}_i 是一个向量

$$\mathbf{w}_i = (w_i^{(1)}, w_i^{(2)}, \dots, w_i^{(n_i)})^T$$

那么求梯度的方法其实就是逐元素求导：

$$\nabla_{\mathbf{w}_i} L = \left(\frac{\partial L}{\partial w_i^{(1)}}, \frac{\partial L}{\partial w_i^{(2)}}, \dots, \frac{\partial L}{\partial w_i^{(n^{(i)})}} \right)^T$$

对矩阵而言是类似的，求梯度的方法依然是逐元素求导：

$$\mathbf{w}_i = \begin{bmatrix} w_i^{(1)(1)} & \dots & w_i^{(1)(n^{(i)})} \\ \vdots & \ddots & \vdots \\ w_i^{(m^{(i)})(1)} & \dots & w_i^{(m^{(i)})(n^{(i)})} \end{bmatrix}$$

$$\Rightarrow \nabla_{\mathbf{w}_i} L = \begin{bmatrix} \frac{\partial L}{\partial w_i^{(1)(1)}} & \dots & \frac{\partial L}{\partial w_i^{(1)(m^{(i)})}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_i^{(n^{(i)})(1)}} & \dots & \frac{\partial L}{\partial w_i^{(n^{(i)})(m^{(i)})}} \end{bmatrix}$$

所以直接求梯度的难度其实并不大，可能就是稍微有些烦琐而已，真正令许多人头疼的其实是梯度的链式法则。虽然它和函数求导的链式法则有些像，但真正用起来时却可能总会感觉到捉襟见肘。下面我们就介绍两个非常实用的计算公式，在大多数情况下，使用这两个公式就已经足够我们解决问题了。

第一个是函数对张量求梯度时的链式法则。假设参数 \mathbf{w}_i 是一个张量， \mathbf{h} 是一个中间变量，且 \mathbf{w}_i 、 \mathbf{h} 、 L 之间满足如下函数关系：

$$\mathbf{h} = f(\mathbf{w}_i), \quad L = g(\mathbf{h})$$

那么就有

$$\nabla_{\mathbf{w}_i} L = \sum_j \frac{\partial L}{\partial h^{(j)}} \cdot \nabla_{\mathbf{w}_i} h^{(j)}$$

其中， j 是 \mathbf{w}_i 和 \mathbf{h} 所有可能的维度，从而 $\frac{\partial L}{\partial h^{(j)}}$ 就是一个普通的函数求导， $\nabla_{\mathbf{w}_i} h^{(j)}$ 就是一个函数直接对张量求梯度的过程，这两个都是我们已经掌握了了的。

下面来看看当 \mathbf{w}_i 是向量和矩阵时链式法则各自的表现是怎样的。假设参数 \mathbf{w}_i 是一个向量， \mathbf{h} 是一个中间变量，且 \mathbf{w}_i 、 \mathbf{h} 、 L 之间满足如下函数关系：

$$h = f(\mathbf{w}_i), \quad L = g(\mathbf{h})$$

其中

$$\mathbf{w}_i = (w_i^{(1)}, w_i^{(2)}, \dots, w_i^{(n_i)})^T, \quad \mathbf{h} = (h^{(1)}, h^{(2)}, \dots, h^{(p)})^T$$

那么就有

$$\nabla_{\mathbf{w}_i} L = \sum_{j=1}^p \frac{\partial L}{\partial h^{(j)}} \cdot \nabla_{\mathbf{w}_i} h^{(j)}$$

而若假设参数 \mathbf{w}_i 是一个矩阵， \mathbf{H} 是一个中间变量，且 \mathbf{w}_i 、 \mathbf{H} 、 L 之间满足如下函数关系：

$$\mathbf{H} = F(\mathbf{w}_i), \quad L = G(\mathbf{H})$$

其中

$$\mathbf{w}_i = \begin{bmatrix} w_i^{(1)(1)} & \cdots & w_i^{(1)(m^{(i)})} \\ \vdots & \ddots & \vdots \\ w_i^{(n^{(i)})(1)} & \cdots & w_i^{(n^{(i)})(m^{(i)})} \end{bmatrix}, \quad \mathbf{h} = \begin{bmatrix} h^{(1)(1)} & \cdots & h^{(1)(q)} \\ \vdots & \ddots & \vdots \\ h^{(p)(1)} & \cdots & h^{(p)(q)} \end{bmatrix}$$

那么就有

$$\nabla_{\mathbf{w}_i} L = \sum_{j=1}^p \sum_{k=1}^q \frac{\partial L}{\partial h^{(j)(k)}} \cdot \nabla_{\mathbf{w}_i} h^{(j)(k)}$$

以上就是第一个重要公式的相关说明。从直观上来讲，它不外乎是把普通的链式法则中的第二项改成了求梯度并把整个链式法则的过程看作各个分量的求和而已。

第二个就是已知某个 element wise 函数因变量的梯度时，求相应自变量的梯度的公式。具体而言，假设参数 \mathbf{w}_i 是一个张量， f 是一个 element wise 函数， $\mathbf{h} = f(\mathbf{w}_i)$ 且 $\nabla_{\mathbf{h}} L$ 已知，那么就有

$$\nabla_{\mathbf{w}_i} L = f'(\mathbf{w}_i) \odot \nabla_{\mathbf{h}} L$$

其中，element wise 函数是指对张量中每个元素都用相同的基函数作用一遍的函数。举个例子，当 f 是一个把 $f^*(x) = 2x$ 作为基函数的 element wise 函数时，假设现在有一个矩阵

$$\mathbf{A} = (a_{ij})_{m \times n} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

那么就有

$$f(\mathbf{A}) = (2a_{ij})_{m \times n} = \begin{bmatrix} 2a_{11} & \cdots & 2a_{1n} \\ \vdots & \ddots & \vdots \\ 2a_{m1} & \cdots & 2a_{mn} \end{bmatrix}$$

然后公式中出现的符号“ \odot ”是 Hadamard Product 的简写，它代表着“逐元素相乘”。具体而言，假设现在有两个矩阵

$$\mathbf{A} = (a_{ij})_{m \times n} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = (b_{ij})_{m \times n} = \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix}$$

那么就有

$$\mathbf{A} \odot \mathbf{B} = (a_{ij}b_{ij})_{m \times n} = \begin{bmatrix} a_{11}b_{11} & \cdots & a_{1n}b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & \cdots & a_{mn}b_{mn} \end{bmatrix}$$

在知道了这两个定义之后我们不难看出，自变量的梯度公式其实可以用链式法则推导出来。以矩阵为例，当参数 \mathbf{w}_i 和中间变量 \mathbf{h} 都是一个 $p \times q$ 的矩阵时，那么就有

$$\nabla_{\mathbf{h}} L = \begin{bmatrix} \frac{\partial L}{\partial h^{(1)(1)}} & \cdots & \frac{\partial L}{\partial h^{(1)(q)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial h^{(p)(1)}} & \cdots & \frac{\partial L}{\partial h^{(p)(q)}} \end{bmatrix}$$

且由 f 是 element wise 函数可知， $\nabla_{\mathbf{w}_i} h^{(j)(k)}$ 是一个除了第 j 行第 k 列的元素非零之外，其余元素都是零的矩阵，且该非零元素即为 $f'(w^{(j)(k)})$ 。

综上所述，即得

$$\nabla_{\mathbf{w}_i} L = \sum_{j=1}^p \sum_{k=1}^q \frac{\partial L}{\partial h^{(j)(k)}} \cdot \nabla_{\mathbf{w}_i} h^{(j)(k)} = \nabla_{\mathbf{h}} L \odot f'(\mathbf{w}_i)$$

换句话说，其实我们只需要链式法则即可在绝大多数情况下完成所有推导，只不过出于方便考虑，我们有时会使用自变量的梯度公式而已。

注意：虽然对于感知机和 SVM 而言，我们只会用到函数对向量的直接求梯度，不过在神经网络中，链式法则就会显得尤为重要。

在有了上面这些求梯度的工具后，我们接下来要关心的就是“更新方向”和“步长（训练速率）”的问题了。由前文的讨论可知，一个自然的想法是令更新方向为负梯度。具体而言，假设参数 \mathbf{w}_i 的更新方向为 $\Delta \mathbf{w}_i$ ，那么我们就可以令

$$\Delta \mathbf{w}_i = -\nabla_{(\mathbf{w}_i)} L$$

而对于训练速率而言，直观的做法就是让它恒等于某个常数 c ：

$$\eta \equiv c$$

其中 η 是我们常用来简记训练速率的符号。这样当然不是最好的做法，不过在实际应用中，这种做法常常也能取得不错的效果。

至此，梯度下降法的关键步骤就已经都被解决了，于是一个梯度下降法的框架就能被自然地写出来了，参见算法 2.5。

算法 2.5 梯度下降法

输入：损失函数 L ，参数 $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_n$ ，迭代次数 M ，训练速率 η ，计算精度 ϵ
过程：

(1) 将参数初始化为随机值

(2) 对 $j = 1, \dots, M$:

(a) 求出参数在当前 L 处的梯度 $\nabla_{\mathbf{w}_i} L$

(b) 取更新方向为负梯度: $\Delta \mathbf{w}_i = -\nabla_{\mathbf{w}_i} L$

(c) 向更新方向 $\Delta \mathbf{w}_i$ 迈出步长为 η 的一步:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \eta \Delta \mathbf{w}_i$$

(d) 若损失函数值的变化或参数的变化小于 ϵ 则退出循环

输出: 经优化的参数 $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_n$

算法 2.5 中的梯度下降法是一个最为朴素的梯度下降法框架, 通过在其基础上结合具体的模型进行改进、拓展能够衍生出一系列著名的算法。具体而言, 这些拓展算法通常会针对如下两个部分进行改进:

- 不是简单地把负梯度作为更新方向, 而是利用更多的(历史)属性来定出更新方向。
- 不单纯地把学习速率设成常量, 而设法让其能够“适应算法”并做出调整。

有关梯度下降的拓展算法会在下一章介绍神经网络模型时进行比较详细的叙述, 这里就不展开叙述了。

至此, 我们对于算法的讨论都是从理论上进行的, 当遇到一个实际问题时, 其实还有一些额外的东西需要考虑。比如对于具体的机器学习模型而言, 我们在训练的时候一般会同时用到许多样本, 此时应用梯度下降法的话就会遇到一个问题: 当计算梯度时, 是应该同时用多个样本进行计算然后求平均, 还是只用单一的样本进行计算? 对这个问题的不同解答, 我们能导出三种不同的实际应用层面的梯度下降法。

- 随机梯度下降法 (Stochastic Gradient Descent, 常简称为 SGD), 它每次都只用单一的样本计算梯度。SGD 的优缺点比较直观, 虽然(在同样的迭代次数下)它的训练速度很快(毕竟只用一个样本计算), 但它搜索解空间的过程会显得比较盲目。在极端情况下, 可能会出现反复来回振荡的行为, 这就导致其最终收敛速度反而可能更慢。如果考虑实际应用的话, 由于 SGD 难以并行实现, 所以其效率往往比较低。
- 小批量梯度下降法 (Mini-Batch Gradient Descent, 常简称为 MBGD), 它每次会从样本集中抽出一部分样本来计算梯度, 通常我们会称这一部分样本为原样本集的一个 Mini-Batch。MBGD 可谓应用得最广泛的梯度下降法, 它在单步迭代中会比 SGD 慢一点, 但它的搜索过程会稳定很多, 从而最终收敛速度一般会比 SGD 快。
- 批量梯度下降法 (Batch Gradient Descent, 常简称为 BGD), 它每次都会用所有的样本来计算梯度。BGD 会有一种“过犹不及”的感觉, 由于在单步迭代中它就会用到所有样本, 所以当训练集很大的时候, 无论是时间开销还是空间开销都会变得让人难以忍受。此外, 用所有样本一起计算梯度虽然能获得很好的统计结果, 但却有可能忽略个体影响, 从而导致更容易陷入整体的局部最优。

联想到代价函数的定义, 上面这三个算法能转述成:

- SGD 想要优化的是单一样本上的代价函数值。
- MBGD 想要优化的是 Mini-Batch 上的代价函数值。
- BGD 想要优化的是整个样本集上的代价函数值。

这其实相当有趣, 我们的最终目的是最小化整个样本集上的代价函数, 但由上述讨论可知, BGD 却往往是最不可行的方法, 正可谓“欲速则不达”。

总之, 现在我们知道了梯度下降法的理论基础和实际应用方法, 所以接下来我们就能着手解决感知机和 SVM 的训练问题了。不失一般性, 我们统一使用 SGD 算法来进行讨论。

先来看感知机, 它的损失函数是

$$L_P(w_0, \mathbf{w}; \mathbf{x}, y) = \frac{1}{\|\mathbf{w}\|} |-y \cdot (w_0 + \mathbf{w}^T \mathbf{x})|_+$$

不难看出

$$\nabla_{\mathbf{w}_i} L_P = \begin{cases} 0 & , y \cdot (w_0 + \mathbf{w}^T \mathbf{x}) \geq 0 \\ \nabla_{\mathbf{w}_i} L_P^* & , y \cdot (w_0 + \mathbf{w}^T \mathbf{x}) < 0 \end{cases}$$

其中

$$L_P^* = -\frac{y}{\|\mathbf{w}\|} (w_0 + \mathbf{w}^T \mathbf{x})$$

由求导和求梯度的公式, 知

$$\begin{aligned} \frac{\partial L_P^*}{\partial w_0} &= -\frac{y}{\|\mathbf{w}\|} \\ \nabla_{\mathbf{w}} L_P^* &= \left(\frac{\partial L_P^*}{\partial w_1}, \frac{\partial L_P^*}{\partial w_2}, \dots, \frac{\partial L_P^*}{\partial w_n} \right)^T \end{aligned}$$

其中

$$\begin{aligned} \frac{\partial L_P^*}{\partial w_i} &= \frac{\partial -\frac{y}{\|\mathbf{w}\|} (w_0 + \sum_{j=1}^n w_j x^{(j)})}{\partial w_i} \\ &= \frac{-y \cdot x^{(i)} \|\mathbf{w}\| + y(w_0 + \mathbf{w}^T \mathbf{x}) \cdot \frac{w_i}{\|\mathbf{w}\|}}{\|\mathbf{w}\|^2} \\ &= -\frac{y}{\|\mathbf{w}\|} \cdot x^{(i)} + \frac{y(w_0 + \mathbf{w}^T \mathbf{x})}{\|\mathbf{w}\|^3} \cdot w_i \end{aligned}$$

于是就有

$$\nabla_{\mathbf{w}} L_P^* = -\frac{y}{\|\mathbf{w}\|} \cdot \mathbf{x} + \frac{y(w_0 + \mathbf{w}^T \mathbf{x})}{\|\mathbf{w}\|^3} \cdot \mathbf{w}$$

从而我们就完成了感知机中各个参数梯度的计算。

对于 SVM 而言，它的代价函数是

$$C_{SVM}(w_0, \mathbf{w}, D) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N |1 - y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

由于代价函数通常是损失函数的均值，所以我们可以倒过来，利用代价函数来求相应的损失函数：

$$L_{SVM}(w_0, \mathbf{w}; \mathbf{x}, y) = \frac{\|\mathbf{w}\|^2}{2} + NC|1 - y \cdot (w_0 + \mathbf{w}^T \mathbf{x})|_+$$

令 $C^* = NC$ ，则

$$L_{SVM}(w_0, \mathbf{w}; \mathbf{x}, y) = \frac{\|\mathbf{w}\|^2}{2} + C^*|1 - y \cdot (w_0 + \mathbf{w}^T \mathbf{x})|_+$$

与感知机类似，我们可以写出

$$\nabla_{\mathbf{w}_i} L_{SVM} = \begin{cases} 0 & , y \cdot (w_0 + \mathbf{w}^T \mathbf{x}) \geq 1 \\ \nabla_{\mathbf{w}_i} L_{SVM}^* & , y \cdot (w_0 + \mathbf{w}^T \mathbf{x}) < 1 \end{cases}$$

其中

$$L_{SVM}^* = \frac{\|\mathbf{w}\|^2}{2} + C^*[1 - y \cdot (w_0 + \mathbf{w}^T \mathbf{x})]$$

由求导和求梯度的公式，知

$$\begin{aligned} \frac{\partial L_{SVM}^*}{\partial w_0} &= w_0 - C^*y \\ \nabla_{\mathbf{w}} L_{SVM}^* &= \left(\frac{\partial L_{SVM}^*}{\partial w_1}, \frac{\partial L_{SVM}^*}{\partial w_2}, \dots, \frac{\partial L_{SVM}^*}{\partial w_n} \right)^T \end{aligned}$$

其中

$$\begin{aligned} \frac{\partial L_{SVM}^*}{\partial \mathbf{w}_i} &= \mathbf{w}_i - \frac{\partial C^*y \cdot (w_0 + \sum_{j=1}^n w_j x^{(j)})}{\partial \mathbf{w}_i} \\ &= \mathbf{w}_i - C^*y \cdot \mathbf{x}^{(i)} \end{aligned}$$

于是就有

$$\nabla_{\mathbf{w}} L_{SVM}^* = \mathbf{w} - C^*y \cdot \mathbf{x}$$

从而我们就完成了 SVM 中各个参数梯度的计算，从而也就完成了所有参数梯度的计算。
总结一下：

$$\begin{cases} \frac{\partial L_P^*}{\partial w_0} = -\frac{y}{\|\mathbf{w}\|} & , \quad \nabla_{\mathbf{w}} L_P^* = -\frac{y}{\|\mathbf{w}\|} \cdot \mathbf{x} + \frac{y(w_0 + \mathbf{w}^T \mathbf{x})}{\|\mathbf{w}\|^3} \cdot \mathbf{w} \\ \frac{\partial L_{SVM}^*}{\partial w_0} = w_0 - C^* y & , \quad \nabla_{\mathbf{w}} L_{SVM}^* = \mathbf{w} - C^* y \cdot \mathbf{x} \end{cases}$$

通过这 4 个公式和算法 2.5, 我们就能完成感知机和 SVM 的训练了。至于具体的代码实现, 笔者利用下一章将会介绍到的、TensorFlow 模型的基本框架 (Base), 实现了 SVM 的原始形式 (LinearSVM) 与 2.3.4 节将会介绍的、应用了核技巧的 SVM 模型。相应的说明会放在附录 A 中, 感兴趣的读者也可以先行参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/b_TraditionalML/SVM.py。

2.3.4* 核技巧

我们在 2.3.1、2.3.2 和 2.3.3 节中进行讨论时, 认为感知机和 SVM 都是线性模型:

$$G(\mathbf{x}) = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$$

在这种情况下, 如果数据集不是线性可分的, 那么感知机将不会收敛, SVM 将会学到一个足够合理但不完美的参数。比如, 它们有可能会在看上去很简单但实际上线性不可分的数据集上表现得不够好, 如图 2.10 和图 2.11 所示。

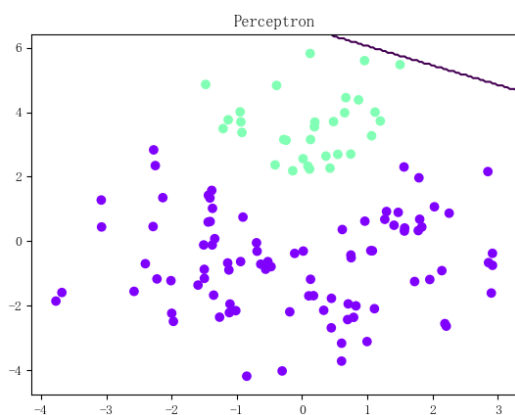


图2.10 感知机在“简单”数据集上的不良表现

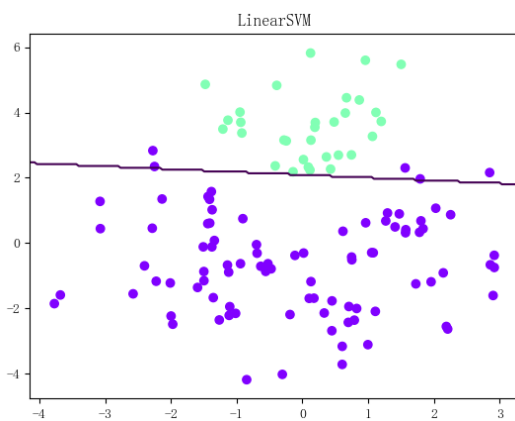


图2.11 SVM在“简单”数据集上的不良表现

用人眼来看其实比较容易看出, 上面这两个数据集可以用一条抛物线来进行完美分类, 但由于目前为止我们讨论的感知机和 SVM 都是线性模型, 所以它们并不能学习出抛物线这种非线性的分界面。其中, 由于数据集线性不可分, 所以感知机直接就发散 (即模型表现不可接受); SVM 虽然也没能学出正确的分界面, 但由于它充分考虑了几何间隔的问题, 所以它得出的分界面在线性分界面中其实已经算是最好的了。

尽管如此, 一个这么“简单”的数据集都不能很好地被解决的话, 作为机器学习模型而言是有所欠缺的。为此我们自然想要对它们加入非线性因素, 而一种常见的、实用的做法就是对它们应用核技巧。由于核技巧虽然很重要但与本书的主题关系不太大, 所以我们将采用直观、

简洁的叙述方式而不追求严谨。

首先来看看核技巧的目标及相应的解决方案是什么。用一句话来直观概括的话，就是：

核技巧会定义一个映射 ϕ 来把特征向量空间映射到更高维的空间中，并期望以此将更高维空间中的线性模型变成原特征向量空间中的非线性模型。

以图 2.10 和图 2.11 对应的数据集为例，虽然它在二维空间（平面）上线性不可分，但如果定义一个映射（假设特征向量为 $\mathbf{x} = (x^{(1)}, x^{(2)})$ ，相应的标签 y 的取值为-1或1）：

$$\phi(x^{(1)}, x^{(2)}) = (x^{(1)}, x^{(2)}, y)$$

即把标签为 1 的特征向量的“高度”设为 1、把标签为-1的特征向量的“高度”设为-1。这样一来，原来在二维空间上线性不可分的数据集，映射到三维空间之后就线性可分了，如图 2.12 所示。

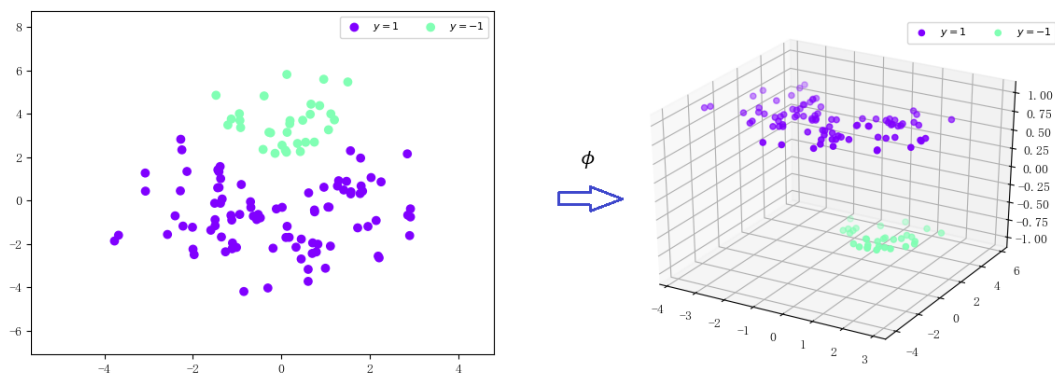


图2.12 从二维空间映射到三维空间，从线性不可分到线性可分

从直观上来说，确实不难想象，同一份数据集在越高维的空间中就越有可能线性可分，但从理论上来说是否真的如此呢？幸运的是，有一个叫 Cover 定理的东西可以保证：对于 N 个点而言，其所在空间的维度越大，则这些点线性可分的概率也就越大。

现在关键点很明确了：我们应该尝试找到合适的 ϕ ，使得一个本来在低维空间中线性不可分的数据集，被 ϕ 映射到高维空间中后能变得线性可分。不过遗憾的是，由于真实数据集的多样性、复杂性都远远超过图 2.12 所示的数据集，直接构造 ϕ 会有大海捞针的无力感。此时核技巧的重要性就体现出来了，它能将构造 ϕ 这个过程转化为构造核函数的过程，而后者通常更为简单、直观。具体而言，核技巧会通过用核函数

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

替换各式算法中出现的内积

$$\mathbf{x}_i^T \mathbf{x}_j$$

来完成将数据集从低维空间映射到高维空间的过程。从直观上来说，之所以这样的替换能够完成从低维到高维的映射，是因为每一种内积都对应着一个空间。其中，最为朴素的内积 $(\mathbf{x}_i^T \mathbf{x}_j)$

对应的是欧式空间，维度一般而言会比较低；而核函数背后的内积（ $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ ）对应的空间的维度往往会很高（比如我们后面马上会介绍到的 RBF 核所对应的空间就是无穷维的），所以用核函数能够将维度提高。

注意：也有用 $\mathbf{x}_i \cdot \mathbf{x}_j$ 来作为内积的简记做法，不过为了统一符号，我们还是使用 $\mathbf{x}_i^T \mathbf{x}_j$ 来作为内积的简记。

而之所以说构造核函数会更为简单直观，是因为有一个叫 Mercer 定理的东西可以保证，只要 $K(\mathbf{x}_i, \mathbf{x}_j)$ 是对称函数，即满足

$$K(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_j, \mathbf{x}_i)$$

的话，那么它具有 Hilbert 空间中内积形式的充分必要条件有以下两个：

- 对任何平方可积函数 g ，都满足

$$\int K(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_i) g(\mathbf{x}_j) d\mathbf{x}_i d\mathbf{x}_j \geq 0$$

- 对含任意 N 个特征向量的数据集 $D = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ ，核矩阵：

$$\mathbf{K} = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \dots & K(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ K(\mathbf{x}_N, \mathbf{x}_1) & \dots & K(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}_{N \times N} = [\mathbf{K}_{ij}]_{N \times N}$$

是半正定矩阵。

Mercer 定理为寻找核函数带来了极大的便利，比如可以证明如下两族函数都是核函数：

- 多项式核

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + 1)^p$$

- 径向基（Radial Basis Function，常简称为 RBF）核

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2\right)$$

以上就是核技巧的大致说明。而在应用到实际问题中时，由于核函数是映射后特征向量之间的内积，所以我们需要把算法中涉及特征向量（ \mathbf{x}_i ）的地方都通过某种变换，变成特征向量之间的内积（ $\mathbf{x}_i^T \mathbf{x}_j$ ）的形式。

综上所述，核技巧的应用步骤大致如下：

- 将算法中出现特征向量的地方，都尝试进行变形并转用特征向量的内积来替代。
- 找到合适的核函数 $K(\mathbf{x}_i, \mathbf{x}_j)$ ，它能返回特征向量 \mathbf{x}_i 、 \mathbf{x}_j 被 ϕ 作用后的内积。
- 用 $K(\mathbf{x}_i, \mathbf{x}_j)$ 替换掉 $\mathbf{x}_i^T \mathbf{x}_j$ ，完成低维到高维的映射，同时也完成了模型从线性到非线性的转换。

在知道了核技巧的思路后，我们就能讨论如何具体地对感知机和 SVM 应用核技巧，从而得到它们的非线性形式了。首先来看感知机，它的代价函数为

$$C_P(G, D) = \frac{1}{N\|\mathbf{w}\|} \sum_{i=1}^N |-y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

而感知机算法的目标是最小化该代价函数，所以我们要让代价函数中的特征向量都变成内积的形式。为此，考虑令

$$\mathbf{w} = \sum_{i=1}^N \alpha_i \mathbf{x}_i$$

则（为了简洁，我们把对 $\|\mathbf{w}\|$ 的转化放在最后）

$$\begin{aligned} C_P(G, D) &= \frac{1}{N\|\mathbf{w}\|} \sum_{i=1}^N \left| -y_i \cdot \left(w_0 + \left(\sum_{j=1}^N \alpha_j \mathbf{x}_j \right)^T \mathbf{x}_i \right) \right|_+ \\ &= \frac{1}{N\|\mathbf{w}\|} \sum_{i=1}^N \left| -y_i \cdot \left(w_0 + \sum_{j=1}^N \alpha_j (\mathbf{x}_j^T \mathbf{x}_i) \right) \right|_+ \end{aligned}$$

在此基础之上应用核技巧是非常普遍的：设核函数为 K ，我们只需要把所有的 $\mathbf{x}_j^T \mathbf{x}_i$ 都换成 $K(\mathbf{x}_j, \mathbf{x}_i)$ 即可：

$$C_P(G, D) = \frac{1}{N\|\mathbf{w}\|} \sum_{i=1}^N \left| -y_i \cdot \left(w_0 + \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) \right) \right|_+$$

这其实就是感知机算法的非线性形式。

注意：除了令 $\mathbf{w} = \sum_{i=1}^N \alpha_i \mathbf{x}_i$ 以外，我们常常能够从理论上推出应该令 $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ （其中 y_i 为-1或1）。这两者的形式都是有一定道理的：由于特征向量空间和参数空间的维度一致（都是 n ），所以当数据集足够好时，我们有理由认为 $\mathbf{x}_1 \sim \mathbf{x}_N$ 能够占满整个特征向量空间，从而它们的线性组合可以表达出同样维度的参数空间中的任意一个参数。换句话说，只要最优的 $\hat{\mathbf{w}}$ 存在，那么就一定存在一组 $\alpha_1 \sim \alpha_N$ ，使得 $\sum_{i=1}^N \alpha_i \mathbf{x}_i = \hat{\mathbf{w}}$ 。然后只需再令 $\alpha_i^* = y_i \alpha_i$ ，就有 $\sum_{i=1}^N \alpha_i^* y_i \mathbf{x}_i = \hat{\mathbf{w}}$ 。

对于 SVM 而言，它的代价函数为

$$C_{SVM}(G, D) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N |1 - y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

用同样的手法不难得出：

$$C_{SVM}(G, D) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N \left| 1 - y_i \cdot \left(w_0 + \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) \right) \right|_+$$

这其实就是 SVM 算法的非线性形式。接下来我们要做的，就是推导 $\|\mathbf{w}\|$ 转化后的形式了。不难由定义得出：

$$\|\mathbf{w}\|^2 = \mathbf{w}^T \mathbf{x} = \left(\sum_{i=1}^N \alpha_i \mathbf{x}_i \right)^T \left(\sum_{i=1}^N \alpha_i \mathbf{x}_i \right) = \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j$$

可见，在出现了特征向量的场合，特征向量都是以内积的形式出现的，所以核技巧确实能够应用在 $\|\mathbf{w}\|$ 上：

$$\|\mathbf{w}\|^2 = \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j)$$

以上，我们就完成了感知机和 SVM 算法的非线性形式的叙述。注意，它们线性形式的模型表达式为

$$G(\mathbf{x}) = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$$

那么在应用了核技巧后，该表达式就应该转化为

$$G(\mathbf{x}) = \text{sign} \left(w_0 + \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) \right)$$

这就是感知机和 SVM 对应的非线性模型。特别对于支持向量机而言，我们在将其简称为“SVM”时，一般指的是非线性模型；对于原始的线性模型，我们常常将其简称为“LinearSVM”。

下面就来看看核技巧的效果。在使用了 RBF 核之后，SVM 在图 2.12 所示的数据集上能有如图 2.13 所示的表现。

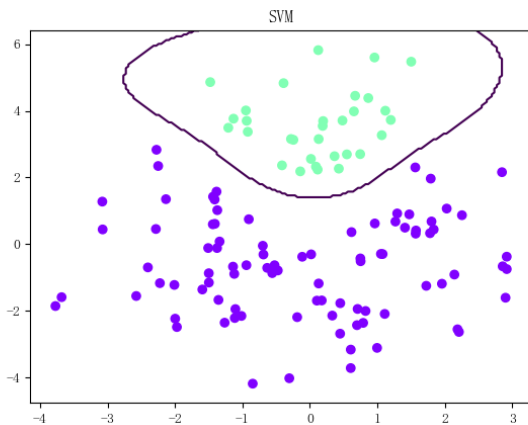


图2.13 SVM在“简单”数据集上的优良表现

可以看到其表现还是相当不错的。

以上，我们就对支持向量机的一些基本概念做了大致的叙述。对于诸如对偶形式、SMO 算法、多分类问题、支持向量回归等知识点，由于它们和本书主题相去甚远，所以我们不在此展开叙述，感兴趣的读者可以参见 <http://www.carefree0910.com/posts/487ba3a6/> 这个系列的文章。

2.3.5 实例演示

我们还是以蘑菇数据集为例，应用 SVM 解决该问题的代码如下所示：

```
01 from sklearn.svm import SVC
02
03 clf = SVC()
04 clf.fit(x_train, y_train)
05 print(np.mean(y_test == clf.predict(x_test)))
```

得到的结果是 0.998587570621，虽然比决策树差一点，但也已经非常不错了。不过需要指出的是，在使用 SVM 解决带有离散型特征的数据时，我们其实仍然需要把输入数据转化为 OneHot Encoding 的数据。而之所以可以在不转化的情况下也能在蘑菇数据集上取得 0.9986 的好结果，一方面是因为 SVM 是一个比较强的模型，另一方面则是因为蘑菇数据集实在是太简单了（附录 D 中会有相应的说明）。

2.4 Logistic 回归

Logistic 回归是挺有意思的一种模型。一来是它的名字有意思，虽然叫作“回归”，但一般而言它却是一个分类模型；二来是它本身很有意思，我们可以赋予它很多数学内涵并把它讲得很复杂，也可以仅从直观上对其进行简单的说明。由于在本书中 Logistic 回归的重要性在于它可以被视为神经网络的基本运算单元（第 3 章会有相应的说明），所以我们就注重于直观地解释它而非细究其背后的数学理论。

Logistic 回归往简单里说，就是感知机的“概率版”——与感知机单纯地进行分类不同，Logistic 回归希望能在感知机的基础上，输出特征向量属于某一类的概率。比如对于二分类问题而言，类别 y 有 1 和 -1 两种取值，感知机模型所希望输出的是具体的分类类别：

$$G(\mathbf{x}) \in \{1, -1\}$$

Logistic 回归则希望输出 $y = 1$ 的概率：

$$G(\mathbf{x}) = p(y = 1) \in (0, 1)$$

我们之前已经说过，感知机模型的形式为：

$$G(\mathbf{x}) = \text{sign}(\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}) \in \{1, -1\}$$

而 Logistic 回归其实就是把上式中 sign 这个取值为 ± 1 的符号函数，换成一个函数取值在 $(0, 1)$ 区间的 σ 函数：

$$G(\mathbf{x}) = \sigma(\mathbf{w}_0 + \mathbf{w}^T \mathbf{x}) \in (0,1)$$

其中 σ 函数通常被称为 Sigmoid 函数，它的定义为：

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

不难看出当 $x \rightarrow -\infty$ 时， $\sigma(x) \rightarrow 0$ ；当 $x \rightarrow +\infty$ 时， $\sigma(x) \rightarrow 1$ 。

Logistic 回归的应用方法已在第 1 章介绍过，这里不再赘述。

注意：与使用 SVM 时类似，在使用 Logistic 回归时，如果数据包含离散型特征，我们仍然需要把输入数据转化为 OneHot Encoding 的数据。而与 SVM 不同的是，在蘑菇数据集上，无论是否进行转化，SVM 都能有接近 100% 的表现；但对于 Logistic 回归来说，转化前的表现只有 95.62% 左右，而转化后的表现则能达到 99.95%，由此可见 OneHot Encoding 的重要性。

2.5 本章小结

- 朴素贝叶斯算法的关键在于条件独立性假设与贝叶斯思维。
- 常见的参数估计有 ML 估计和 MAP 估计两种，其中 MAP 估计比 ML 估计多了对数先验概率 $\log p(\theta)$ 这一项，体现了贝叶斯学派的思想。
- scikit-learn 的 MultinomialNB 模型要求输入数据是 OneHot Encoding 的数据。
- 决策树对应着一个决策路径，它会利用各个特征把特征向量空间划分为许多子空间，然后再利用某些准则给这些子空间打上标签。
- 决策树常用的剪枝算法有两种（预剪枝、后剪枝），它们都是为了适当地降低模型复杂度，从而期望模型在未知数据上的表现更好。
- 感知机只能保证对线性可分数据集进行正确分类，但它没有过多考虑泛化能力。
- SVM 通过考虑优化几何间隔来增强模型的泛化能力。
- 核技巧能够将线性算法“升级”为非线性算法。
- Logistic 回归可以看作感知机模型的“概率版”，它将符号函数 sign 换成了一个函数取值在(0,1)区间的 σ 函数（Sigmoid 函数），从而使得模型具有了概率意义。
- 虽然决策树、SVM、Logistic 回归都不会硬性要求对输入数据中的离散型特征做 OneHot Encoding 转化，但是只有对离散型特征做了 OneHot Encoding 之后，这些模型的表现才是“正常”的。换句话说，如果把原始的、只进行过数值化操作的数据输入模型的话，模型的表现其实是有点“病态”的。

第 3 章

神经网络入门

我们在第 2 章介绍了 5 种比较传统且经典的机器学习算法，在过去的很长一段时间里，这 5 种算法及其变种有着非常广泛且有效的应用。其中，虽然决策树的原始模型可能稍显逊色，但是基于决策树的一系列集成算法，比如随机森林（Random Forest）和 xgboost，它们直至今日在解决结构化数据问题上都有着很大的优势。

本章我们将介绍的就是本书的主题——神经网络（Neural Network，常简称为 NN）的一些基本知识。正如第 1 章开头所说，人工智能领域中最为火热且行之有效的技术就是深度学习，而神经网络正是深度学习的根基。

不过，尽管神经网络的原始算法的提出已经是比较久远的事情，相应的数学理论也提出了不少，许多人也认为它和第 2 章介绍的那些传统机器学习算法一样应该归于统计学习方法，不过我们听到的比较多的说法是：神经网络是一个性能优异的“黑箱”，我们发现它的效果非常棒，但它到底在模型内部做了什么却很难解释清楚。所以对于本书而言，我们希望能够给大家提供一些神经网络之所以强大的直观感受，同时会介绍一些能让神经网络表现得更好的方法和技巧。我们希望通过本书让大家对学习神经网络更有自信，不会因为它是一个“黑箱”而心存畏惧。

为此，本章将先介绍神经网络的基本原理，然后在下一章用神经网络与第 2 章所介绍过的传统算法进行对比，并由此显示神经网络的强大。从第 5 章开始，我们会在最朴素的神经网络的基础上做一些拓展，这些拓展能够使我们的网络结构适应于更广泛的数据。

此外，正如前言所说，本书是偏工程化的应用类书籍，所以我们会在 3.5 节给出一个比较完整的 TensorFlow 模型的基本框架的实现。在这个框架的基础上，实现朴素神经网络（BasicNN）、增强版神经网络（AdvancedNN）、循环神经网络（RNN）、卷积神经网络（CNN）等结构都是非常轻松且高效的。事实上，我们只需把模型的主要结构写出来，就能共享基本框架所带来的评估系统、保存复用系统和超参数管理系统等，这就能让我们专注于建模而不必操心于各种琐

碎的实现细节。

本章主要涉及的知识点有：

- 神经网络的基本运算单元与组合方式
- 神经网络的激活函数与损失函数
- 神经网络的训练与参数更新方法
- TensorFlow 模型基本框架的搭建
- 神经网络的朴素实现与实际应用

3.1 神经网络的结构

在 2.4 节我们曾经说过，Logistic 回归可以视为神经网络的基本运算单元。当时我们将 Logistic 回归看作一个单独的分类器，它的模型的表达式为

$$G(\mathbf{x}) = \sigma(\mathbf{w}_0 + \mathbf{w}^T \mathbf{x})$$

如果想将它拓展成更一般的形式，我们不必将输出拘泥为一个 0 到 1 之间的数。具体而言，当把 Logistic 回归视为一个基本运算单元时，它的表达式就可以是

$$f(\mathbf{x}) = \phi(\mathbf{w}_0 + \mathbf{w}^T \mathbf{x})$$

其中， ϕ 是我们下一节将会进行介绍的“激活函数（Activation Function）”。由于我们的模型叫“神经网络”，所以通常会称这个基本运算单元为“神经元（Neuron）”。

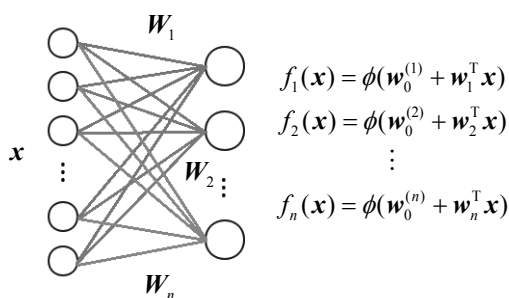
在将 Logistic 回归视为一个基本运算单元（神经元）后，我们就能将 Logistic 回归进行“堆叠”，从而形成一个更大、更复杂的模型。在神经网络中，我们会将很多 Logistic 回归对应的神经元组成一个“层（Layer）”结构，然后将很多个层依次排开，并使用矩阵运算来作为层与层之间的联系。具体而言，神经网络会做这样两件事：

- 将许多神经元组成模型的一个层结构，如图 3.1 所示。由于这些神经元已经被视为一个整体，所以我们应该把相应的表达式也进行一个整合：

$$\mathbf{f}(\mathbf{x}) \triangleq (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_n(\mathbf{x}))^T = \phi(\mathbf{W}_0 + \mathbf{W}\mathbf{x})$$

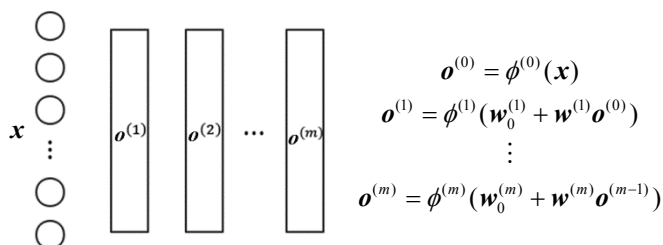
其中

$$\mathbf{W}_0 = (\mathbf{w}_0^{(1)}, \mathbf{w}_0^{(2)}, \dots, \mathbf{w}_0^{(n)})^T, \quad \mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_n^T \end{bmatrix}$$

图3.1 许多神经元 (f_1, f_2, \dots, f_n) 组成了一个层结构 (x 是输入)

注意：在实际应用中，我们默认将同一层的所有激活函数设为同一个激活函数，而不会对同一层的每个神经元都分配一个独立的激活函数。这是因为如果不统一激活函数的话，虽然理论上的最优解可能会比统一激活函数要好一点，但和带来的实现上的不便与运行速度的折损相比，这点优势就会被抵消，甚至得不偿失。

- 将许多层结构视为整体并作为最终的神经网络模型，如图 3.2 所示。在实际应用中，我们一般会在神经网络中用许许多多的层（即图中所示的 n 比较大），于是我们的神经网络就会比较“深”。这也是“深度学习”中“深度”二字的来由之一

图3.2 许多层结构 ($o^{(0)}, o^{(1)}, \dots, o^{(m)}$) 组成了神经网络

注意：在第 1 章我们曾经说过，在把原始输入送进机器学习模型之前，一般会需要先做数据预处理。在神经网络中，我们可以把数据预处理视为一个层结构的内部操作，所以我们一般会认为模型的输入 (x) 是一个叫“输入层”的层结构的输出 ($o^{(0)}$)。再加上第 m 层 ($o^{(m)}$) 其实就是我们神经网络模型的最终输出，所以就可以发现神经网络的一个有趣的性质：输入、输出以及中间的所有运算都会由层结构来完成。

看到这里可能会有读者想问：为什么神经网络的第一步要将许多神经元组成一个层结构？直接随意地用这些神经元构成一个神经网络不行吗？事实上从理论的角度来说，不将神经元限制为层结构反而是更好的做法，因为它拥有更高的自由度。也就是说，理想的神经网络模型其实应该是一个类似于图 3.3 所示的有向无环图 (DAG 图)：

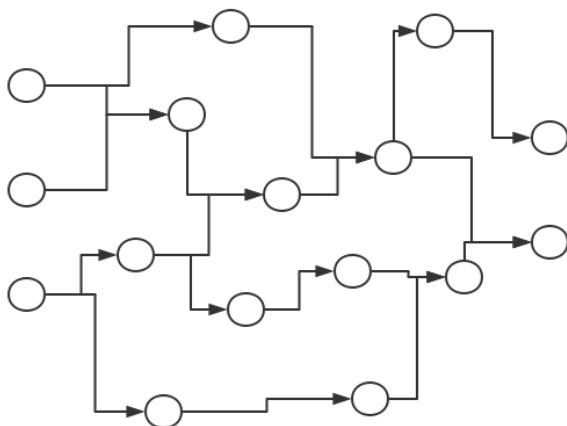


图3.3 一个输入为三维、输出为二维的DAG图

如果神经网络模型真的能够对任意由神经元组成的 DAG 图进行高效训练的话，那么说它能够和真正的、我们人类的神经网络类比可能也不算夸张。然而遗憾的是，由于现在我们对矩阵运算的依赖程度很大（因为矩阵运算是被高度优化了的），所以目前主流的神经网络模型结构基本都是像我们之前所说的那样，需要先把神经元组成一个层结构，然后再用层结构来组合出最终的神经网络模型。

由上述讨论可以看出，对于神经网络而言，它的超参数至少有如下三种：

- 层结构的个数是 $(m + 1)$ 。
- 每个层结构中神经元的个数是 $(n^{(i)})$ 。
- 每个层结构所对应的激活函数是 $(\phi^{(i)})$ 。

参数则至少有如下两种：

- 每一层的偏置量 $(\mathbf{w}_0^{(i)})$ 。
- 层与层之间的权值矩阵 $(\mathbf{w}^{(i)})$ 。

随着讨论的深入，我们会发现神经网络还有各式各样的超参数与参数，不过上面这 4 个是基本中的基本，需要大家熟记在心。

3.2 前向传导算法

神经网络的诸多算法比起一些经典的传统机器学习算法（比如第 2 章所介绍的 4 种机器学习算法）而言确实会显得更为复杂，再加上现在的许多机器学习算法库已经在底层把所有的算法细节都考虑得很详细并给出了高级的 API 以供调用，所以现在有大量能够熟练运用神经网络解决问题、却对其基本算法不甚了解的人。不过笔者认为，虽然我们确实不必将每个理论细节都钻研透彻，但对于一些基本的概念和推导却还是有必要知道的。为此，我们将会在本节中叙述神经网络中比较简单的算法——前向传导算法，它是神经网络进行输出的基础；至于神经网络中比较令人头疼的“反向传播算法”，我们会在下一节中进行介绍，它是神经网络进行训练的基础。

3.2.1 算法概述

我们在 3.1 节曾说过，现在的神经网络模型依赖于矩阵运算并总会先将神经元组合成层结构，而且模型的输入、输出以及中间的所有运算都会由层结构来完成，所以神经网络模型的基本组成单元并不是神经元而应该是层。本节的讨论都将会以层为基本单位展开叙述，这其实也能帮助大家尽快适应矩阵运算的各种方法。

神经网络的结构在图 3.2 中已经得到了很好的展现，这里我们就以符号语言重新叙述一遍。假设我们原始输入的特征向量为 \mathbf{x} ，神经网络一共有 $m+1$ 个层结构，那么输入、输出和这 $m+1$ 个层结构之间的关系就可以归纳为如下三点：

- 输入层接收 \mathbf{x} 后，在内部进行数据预处理并把结果传给第一个隐藏层：

$$\mathbf{o}^{(0)} \leftarrow \phi^{(0)}(\mathbf{x})$$

其中 $\phi^{(0)}$ 为数据预处理对应的函数。数据预处理相关的比较详细的说明会在第 6 章中进行，它的一个简明有效的定义方式则会在 3.2.3 节中给出。

- 模型的输出即为第 m 层的输出：

$$G(\mathbf{x}) = \mathbf{o}^{(m)}(\mathbf{x}) = \phi^{(m)}(\mathbf{W}_0^{(m)} + \mathbf{W}^{(m)}\mathbf{o}^{(m-1)}) = \dots$$

其中我们通常会称 $\phi^{(m)}$ 为“变换函数”，这是因为我们通常会需要模型输出一个概率向量，那么我们就可以利用 $\phi^{(m)}$ 来将输出层的输入值 $\mathbf{W}_0^{(m)} + \mathbf{W}^{(m)}\mathbf{o}^{(m-1)}$ “变换”成一个概率输出（对于 $\mathbf{W}_0^{(m)} + \mathbf{W}^{(m)}\mathbf{o}^{(m-1)}$ ，则通常称它为模型的“原始输出”）。

- 对于模型的第 1 层（输入层是第 0 层）到第 $m-1$ 层，我们通常会称它们为“隐藏层（Hidden Layer）”，它们的作用是将输入层非线性映射到输出层，是整个神经网络模型强大的表达能力的根本来源。

注意：我们在计算神经网络的层数时，常常不会把输入层计算在内。换句话说，图 3.2 所示的神经网络就是一个“ m 层神经网络”。

于是神经网络中所谓的“前向传导算法（Forward Propagation）”就可以总结为如下三步。这些步骤在前文的诸多叙述中已经有所体现，这里只是做一个整理：

- 输入层将原始输入进行数据预处理并将结果输出

$$\mathbf{o}^{(0)} = \phi^{(0)}(\mathbf{x})$$

- 每个隐藏层都会把上一层的输出做一个线性映射后当成自己的输入。具体而言，假设当前层为第 i 层，那么它接收的输入 $\mathbf{h}^{(i)}$ 即为

$$\mathbf{h}^{(i)} = \mathbf{W}_0^{(i)} + \mathbf{W}^{(i)}\mathbf{o}^{(i-1)}$$

然后在层结构的内部，隐藏层一般会使用激活函数来作用于该输入并得到输出 $\mathbf{o}^{(i)}$

$$\mathbf{o}^{(i)} = \phi^{(i)}(\mathbf{h}^{(i)})$$

- 对于输出层而言，则是通过变换函数来将模型的原始输出 $\mathbf{h}^{(m)}$ 变换成最终输出 $\mathbf{o}^{(m)}$

$$G(\mathbf{x}) = \mathbf{o}^{(m)} = \phi^{(m)}(\mathbf{h}^{(m)}) = \phi^{(m)}(\mathbf{W}_0^{(m)} + \mathbf{W}^{(m)}\mathbf{o}^{(m-1)})$$

而在第 1 章、第 2 章我们都曾经提到过，神经网络需要定义一个损失函数，然后通过计算损失函数对各个参数的梯度并利用梯度下降来完成模型的训练。所以除了上述三个步骤以外，完整的前向传导过程还应该包括如下这样一步：

- 根据事先定义好的损失函数 L ，利用模型输出和相应的真实标签来算出模型的损失。

以上就是前向传导算法的全部过程，接下来我们就来具体地操作一下以加深理解。用一个简单的神经网络结构为例，如图 3.4 所示。

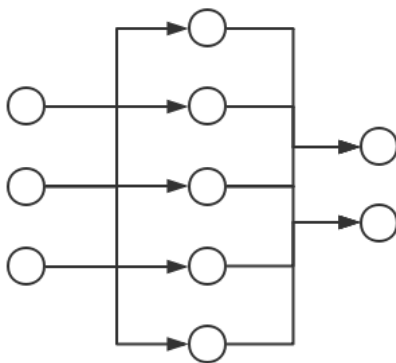


图 3.4 一个简单的两层（单隐含层）神经网络

注意：虽然图 3.4 将一个个神经元画了出来，但是在本节及今后的所有讨论中，我们都应该时刻记住：神经网络的基本组成单元是层（Layer）而不是神经元，这里用了神经元仅仅是因为我们刚起步，需要一个过渡的过程。

在展开叙述前，做一些符号约定是有必要的。这些符号约定非常重要，大多数在前文也已有提及，这里只是做了汇总以方便查阅。

- 记图 3.4 中的神经网络从左到右对应的 Layer 为 L_0 、 L_1 、 L_2 ，记 L_i 中从上往下数的第 j 个神经元为 u_{ij} 。
- 记 L_i 对应的：
 - 神经元个数为 n_i （从而 $n_0 = 3$ 、 $n_1 = 5$ 、 $n_2 = 2$ ）。
 - 激活函数、偏置量分别为 ϕ_i 、 $\mathbf{w}_0^{(i)}$ 。
- 记 L_{i-1} 和 L_i 之间的权值矩阵为 $\mathbf{W}^{(i)}$ 、神经元 $u_{i-1,j}$ 和 $u_{i,k}$ 之间的权值为 $w_{jk}^{(i)}$ ，可知：

$$\mathbf{W}^{(i)} = \begin{bmatrix} w_{11}^{(i)} & \cdots & w_{1,n_{i-1}}^{(i)} \\ \vdots & \ddots & \vdots \\ w_{n_i 1}^{(i)} & \cdots & w_{n_i, n_{i-1}}^{(i)} \end{bmatrix}_{n_i \times n_{i-1}}, \quad i = 1, 2$$

- 记 L_i 对应的输入为 $\mathbf{h}^{(i)}$ 、输出（或说激活值）为 $\mathbf{o}^{(i)}$ 。
- 记模型的输入为特征向量 \mathbf{x} ，对应的标签为 y ，同时记损失函数为 L 。除非是特殊的情况，一般而言我们会要求 L 是一个二元对称函数，即

$$L(G(\mathbf{x}), y) = L(y, G(\mathbf{x}))$$

那么上述神经网络的前向传导算法的所有步骤即为：

- 在输入层中进行数据预处理

$$\mathbf{o}^{(0)} = \phi^{(0)}(\mathbf{x})$$

这里的 $\phi^{(0)}$ 代指数据预处理函数，注意不管 \mathbf{x} 的维度是多少，由于图 3.4 所示的神经网络的输入层只有 3 个神经元，所以 $\mathbf{o}^{(0)}$ 都需要是一个 3 维的列向量（ $\mathbf{o}^{(0)} \in \mathbb{R}^3$ ）。

- 根据输入层的输出，算出隐藏层的输入和输出

$$\begin{aligned}\mathbf{h}^{(1)} &= \mathbf{W}_0^{(1)} + \mathbf{W}^{(1)}\mathbf{o}^{(0)} \\ \mathbf{o}^{(1)} &= \phi^{(1)}(\mathbf{h}^{(1)})\end{aligned}$$

注意由于 $\mathbf{o}^{(0)} \in \mathbb{R}^3$ 、 $\mathbf{o}^{(1)}$ 是 5 维的列向量（ $\mathbf{o}^{(1)} \in \mathbb{R}^5$ ），所以偏置量 $\mathbf{W}_0^{(1)}$ 是 5 维的列向量，权值矩阵 $\mathbf{W}^{(1)}$ 是一个 5×3 的矩阵。

- 根据隐藏层的输出，算出输出层的输入和输出

$$\begin{aligned}\mathbf{h}^{(2)} &= \mathbf{W}_0^{(2)} + \mathbf{W}^{(2)}\mathbf{o}^{(1)} \\ \mathbf{o}^{(2)} &= \phi_2(\mathbf{h}^{(2)})\end{aligned}$$

这里的 $\phi^{(2)}$ 代指变换函数。注意由于 $\mathbf{o}^{(1)} \in \mathbb{R}^5$ 、 $\mathbf{o}^{(2)}$ 是 2 维的列向量（ $\mathbf{o}^{(2)} \in \mathbb{R}^2$ ），所以偏置量 $\mathbf{W}_0^{(2)}$ 是 2 维的列向量，权值矩阵 $\mathbf{W}^{(2)}$ 是一个 2×5 的矩阵。

- $\mathbf{o}^{(2)}$ 即为模型的输出， $L(\mathbf{o}^{(2)}, y) = L(y, \mathbf{o}^{(2)})$ 即为模型在样本 (\mathbf{x}, y) 上的损失。

注意：以上这个例子中的神经网络模型其实是一个二分类模型（ $n_3 = 2$ ），如果想用神经网络解决多分类问题（比如 K 分类问题），只需自然地将输出层的神经元个数设为类别个数（ $n_2 \leftarrow K$ ）即可，此时我们要求标签 y 是（ K 维的）OneHot 标签。

至此，我们就对前向传导算法有了一个大致认知。在接下来的章节中，我们会先在 3.2.2 节里介绍一下前向传导算法中各个步骤的直观内涵，然后对于上文遗留的一些没有说明的细节（比如如何为激活函数、变换函数，以及损失函数有哪些选择等），会在 3.2.3 和 3.2.4 节中逐一讨论。

3.2.2 算法内涵

由 3.2.1 节的讨论不难总结出，前向传导算法的核心在于如下两个运算步骤：

- 层与层之间的线性映射。
- 层结构内部激活函数的使用。

其中，层与层之间的线性映射将由矩阵运算 $\mathbf{h}^{(i)} = \mathbf{W}_0^{(i)} + \mathbf{W}^{(i)}\mathbf{o}^{(i-1)}$ 来完成。在这个步骤中，权值矩阵可以将特征向量从某个维度（ $n^{(i-1)}$ ）线性映射到另一个维度（ $n^{(i)}$ ），它的重要性似乎

比较明显。一方面，如果 $n^{(i-1)} < n^{(i)}$ 的话，注意我们在 2.3 节介绍 SVM 时曾经提过，越高维的空间中， N 个点线性可分的概率就越高，这意味着映射所得的 $\mathbf{h}^{(i)}$ 比原始的 $\mathbf{o}^{(i-1)}$ 更有潜力。再加上矩阵运算是被高度优化过的，所以权值矩阵从理论和实际的角度来看都不可或缺。

不过与 SVM 中的核函数有所不同的是，线性映射（顾名思义）是线性的，而核函数则是非线性的，这一点上的差距将直接凸显出第二步“激活函数的使用”的重要性。事实上，如果我们不使用激活函数的话，那么不难看出

$$\begin{aligned} G(\mathbf{x}) &= \mathbf{W}_0^{(m)} + \mathbf{W}^{(m)} \mathbf{o}^{(m-1)} \\ &= \mathbf{W}_0^{(m)} + \mathbf{W}^{(m)} (\mathbf{W}_0^{(m-1)} + \mathbf{W}^{(m-1)} \mathbf{o}^{(m-2)}) \\ &\triangleq \widetilde{\mathbf{W}}_0^{(m)} + \widetilde{\mathbf{W}}^{(m)} \mathbf{o}^{(m-2)} \end{aligned}$$

其中

$$\begin{aligned} \widetilde{\mathbf{W}}_0^{(m)} &= \mathbf{W}_0^{(m)} + \mathbf{W}^{(m)} \mathbf{W}_0^{(m-1)} \\ \widetilde{\mathbf{W}}^{(m)} &= \mathbf{W}^{(m)} \mathbf{W}^{(m-1)} \end{aligned}$$

依此类推，不难得出我们的模型表达式最终会退化成

$$G(\mathbf{x}) = \widetilde{\mathbf{W}}_0^{(1)} + \widetilde{\mathbf{W}}^{(m)} \mathbf{x}$$

即此时神经网络就退化成一个线性模型，而这也是我们常说激活函数是神经网络非线性性来源的原因。

所以权值矩阵 $\mathbf{W}^{(i)}$ 和激活函数 $\phi^{(i)}$ 的重要性都有点“不言而喻”的感觉，但是线性映射中的另外一项——偏置量 $\mathbf{W}_0^{(i)}$ 的重要性相对而言可能没那么明显。为了直观体会偏置量 $\mathbf{W}_0^{(i)}$ 的重要性，可以设想这样一个场景（仍取图 3.4 中的结构来说明问题）：

- 激活函数全是中心对称的函数（比如常见的 Tanh 函数），即：

$$\phi^{(i)}(\mathbf{x}) + \phi^{(i)}(-\mathbf{x}) = 0, \quad i = 1, 2$$

- 训练样本集为：

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2)\}$$

其中

$$\mathbf{x}_1 = (-1, -1, -1)^T, \quad y_1 = -1$$

$$\mathbf{x}_2 = (+1, +1, +1)^T, \quad y_2 = -1$$

- 权值矩阵 $\mathbf{W}^{(1)}$ 、 $\mathbf{W}^{(2)}$ 可变，但偏置量 $\mathbf{W}_0^{(1)} \equiv \mathbf{W}_0^{(2)} \equiv \mathbf{0}$ 。

在此场景下就可以证明，无论怎样进行训练，模型 G 在训练集 D 上的准确率都不可能达到 100%。这是因为我们总会有：

$$G(\mathbf{x}) = \phi^{(2)} \left(\mathbf{W}^{(2)} \phi^{(1)} (\mathbf{W}^{(1)} \mathbf{x}) \right)$$

从而由激活函数为中心对称函数可知：

$$\begin{aligned} G(-\mathbf{x}) &= \phi^{(2)}\left(\mathbf{W}^{(2)}\phi^{(1)}\left(\mathbf{W}^{(1)}(-\mathbf{x})\right)\right) = \phi^{(2)}\left(\mathbf{W}^{(2)}\phi^{(1)}(-\mathbf{W}^{(1)}\mathbf{x})\right) \\ &= \phi^{(2)}\left(\mathbf{W}^{(2)}\left(-\phi^{(1)}(\mathbf{W}^{(1)}\mathbf{x})\right)\right) = \phi^{(2)}\left(-\mathbf{W}^{(2)}\phi^{(1)}(\mathbf{W}^{(1)}\mathbf{x})\right) \\ &= -\phi^{(2)}\left(\mathbf{W}^{(2)}\phi^{(1)}(\mathbf{W}^{(1)}\mathbf{x})\right) = -G(\mathbf{x}) \end{aligned}$$

即

$$G(\mathbf{x}_1) = -G(\mathbf{x}_2)$$

但我们有 $y_1 = y_2 = -1$ ，所以模型 G 不可能同时预测对 (\mathbf{x}_1, y_1) 和 (\mathbf{x}_2, y_2) 。事实上由上述讨论可知，此时模型 G 所做的预测必定是关于输入空间“中心对称”的，这当然不是一个良好的结果。而如果我们引入偏置量的话，上述的对称性就会被打破，这就是偏置量 $\mathbf{W}_0^{(i)}$ 重要性的其中一个比较浅显、直观的方面。

将上述讨论总结一下，可以归纳出如下三个要点：

- 权值矩阵 $\mathbf{W}^{(i)}$ 能将数据线性映射到高维空间中。
- 偏置量 $\mathbf{W}_0^{(i)}$ 能打破模型的对称性。
- 激活函数 $\phi^{(i)}$ 能提供非线性性。

3.2.3 激活函数

本节我们将简要介绍一下前文不断提及却又没有细说的激活函数 ϕ 。激活函数的英文叫 Activation Function，是神经网络中最重要的元素之一。正如 3.2.2 节所说的，它是整个神经网络中的非线性扭曲力。如果没有激活函数（或说激活函数是恒同映射 $\phi(x) = x$ ）的话，神经网络其实就只是一个线性模型而已。

那么既然激活函数那么重要，是不是只有特定的一类函数才能作为激活函数？又应该如何选择合适的激活函数呢？对于第一个问题的答案，Universal approximation theorem（万能逼近定理）告诉我们，只要我们选择非常值的、单调递增的、有界连续函数作为激活函数，那么神经网络从理论上就能逼近任意连续函数。对于第二个问题，一般而言我们都从经验的角度出发进行选择，而且这些经验并不能保证通用性。换句话说，并没有一个理论告诉我们应该如何选择，而只有一些行之有效的方法帮助我们更快地找到理想的激活函数。

注意：虽然 Universal approximation theorem 这个理论保证了逼近任意连续函数的神经网络的存在性，但它并没有保证可学习性，也就是说，实际上可能根本不可能训练出那个理论上存在的神经网络。再加上近期的研究结果发现，即使是无界的激活函数（比如 ReLU）也能达成这种性质，所以现在有许多新的激活函数（比如接下来会介绍到的、近期由谷歌大脑 Google Brain 提出的 Swish 函数就不是单调递增的）并不严格遵循“非常值、单调递增、有界连续”这些条件。

下面就来具体介绍几个常见的激活函数，在实际应用中，如果没有特殊的先验知识，我们

一般都会在下面这些激活函数中做出选择。

- 逻辑函数 (Sigmoid)

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

其函数图像如图 3.5 所示。Sigmoid 函数正是 Logistic 回归中用的变换函数，其优缺点比较鲜明：它是非常光滑的函数，在生物学意义上和神经元的激活方式（阶跃式激活）很相近，但由于它两端很平坦，即相应的导数会趋于 0，从而就很容易产生“梯度消失”的问题（梯度消失的内涵会在后文介绍）。

- 正切函数 (Tanh)

$$\phi(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

其函数图像如图 3.6 所示。由图 3.5 可知，Sigmoid 函数是无法输出负值的，且函数本身不具有对称性，这在某些问题上是一个不良的性质，而 Tanh 解决了 Sigmoid 无法输出负值和不对称的问题。不过由于其函数两端仍然趋于平坦，所以梯度消失的问题仍然存在。

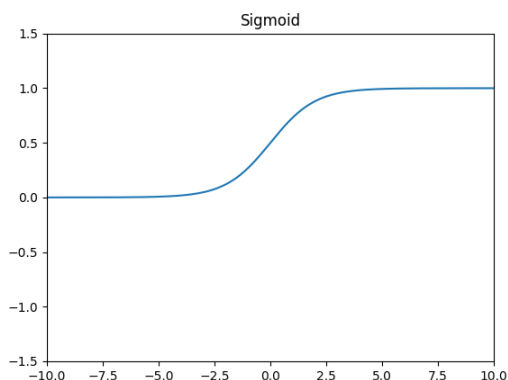


图3.5 Sigmoid函数的图像

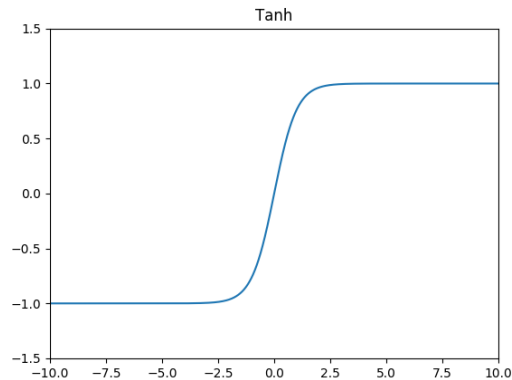


图3.6 Tanh函数的图像

- 线性整流函数 (Rectified Linear Unit, 常简称为 ReLU)

$$\phi(x) = \max(0, x)$$

其函数图像如图 3.7 所示，注意纵轴范围与上述两个激活函数不同。ReLU 是一个非常有意思的激活函数，它的形式简洁、求导方便，在大于 0 的地方不会存在梯度消失的现象，但同时它也无法输出负值，且函数也不具有对称性。然而实践证明，ReLU 在绝大多数情况下都是一个非常好的选择，即无法输出负值、不具有对称性在绝大多数情况下并不构成缺陷。这其实有一些反直观，但仔细一想又不无道理：我们人类的神经元其实也是这样的，在不被激活时都是“沉寂”的，即输出的信号是 0，只有在被激活时才会输出相应的值。此外，ReLU 在 0 点处其实是不可导的，然而一方面是输入恰好为 0 的概率通常为 0，另一方面则是简单地令其在 0 点处导数为 0 也不会给反向传播算法带来太大的麻烦。

- Leaky ReLU 函数

$$\phi(\alpha, x) = \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

其函数图像如图 3.8 所示（图 3.8 中的函数图像对应的 α 取值为 0.01）。Leaky ReLU 函数可以看作 ReLU 的一个“下穿”版本，它在大于 0 的情况下和 ReLU 表现一致，在小于 0 时则可以输出（线性的）负值，且它在 0 点仍然不可导。

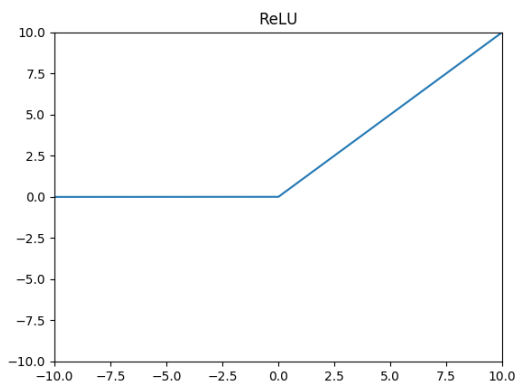


图3.7 ReLU函数的图像

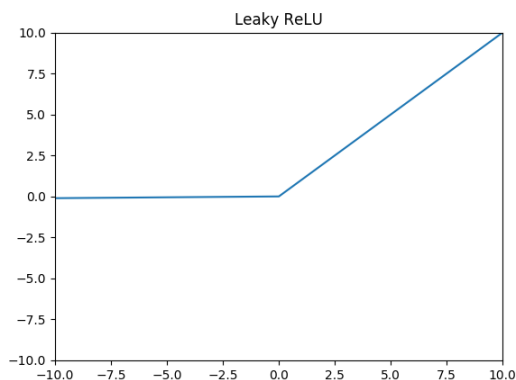


图3.8 Leaky ReLU函数的图像

- ELU 函数（Exponential Linear Unit）

$$\phi(\alpha, x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

其函数图像如图 3.9 所示，图 3.9 中的函数图像对应的 α 取值为 1。ELU 函数可以看作 ReLU 的一个“平滑下穿”版本，它在大于 0 的情况下和 ReLU 表现一致，在小于 0 时则可以输出（非线性的）负值。此外，当 ELU 中的 α 取值为 1 时，它在 0 点处是可导的。

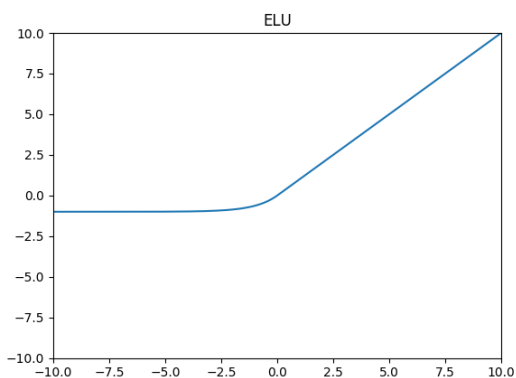


图3.9 ELU函数的图像

- SELU 函数

$$\phi(x) = 1.0507 \times \begin{cases} 1.6733(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

其函数图像如图 3.10 所示。SELU 函数可以看作 ELU 函数的一个特殊版本，它那两个超参数的选取（1.0507 和 1.6733）并非无中生有，而是有一整套完整的理论作为支撑的。笔者囿于能力，并不能将原论文读完，所以我们仅会在本节的最后简单地介绍一下 SELU 函数，感兴趣的读者可以参见 <https://arxiv.org/pdf/1706.02515.pdf> 以了解更多的细节。

- Softplus 函数

$$\phi(x) = \ln(1 + e^x)$$

其函数图像如图 3.11 所示。Softplus 可以看作 ReLU 的一个“平滑”版本，它在输入足够大的情况下和 ReLU 的表现基本一致，在小于 0 的情况下仍然不能输出负值，不过它在 0 点是可导的。

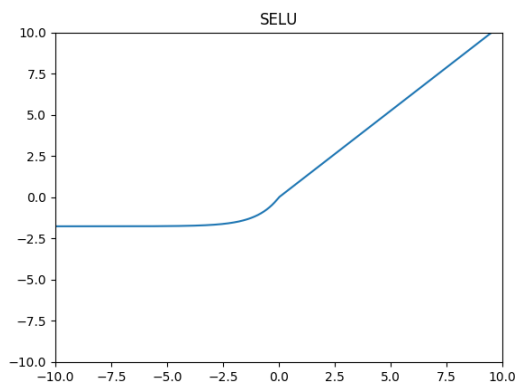


图3.10 SELU函数的图像

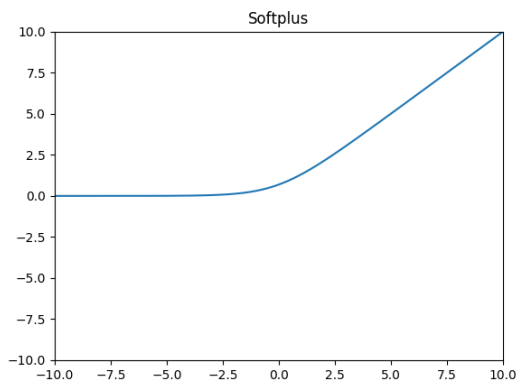


图3.11 Softplus函数的图像

- 恒同映射（Identity）

$$\phi(x) = x$$

其函数图像从略。

- Swish 函数

$$\phi(x) = \frac{x}{1 + e^{-x}}$$

其函数图像如图 3.12 所示。Swish 函数是恒同映射和 Sigmoid 函数的乘积，其发明者（Google Brain）声称它能在大多数情况下直接取代 ReLU 并获得更好的效果。不过由于相关的研究还有一定的争议性，这里就不详细展开讨论了。

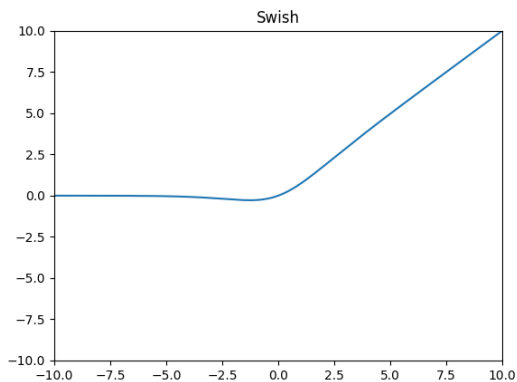


图3.12 Swish函数的图像

以上就是到目前（2017 年 11 月）为止比较常见的激活函数。我们在前文曾经说过，如何从这些函数中选出合适的函数是一个经验性的过程，这里我们就引用 cs231n 上的一个选择方法：“总之先选择 ReLU，为此你需要仔细调整训练速率以防止太多的神经元‘死亡’（因为 ReLU 在小于 0 的情况下输出恒为 0，所以很有可能就一直卡在恒为 0 的情况，于是它就‘死亡’了）。如果你觉得这两个问题很糟心，那么可以试试 Leaky ReLU。不过需要注意的是，永远不要用 Sigmoid，如果你真的特别想用 Sigmoid 的话也得用 Tanh 来代替它，不过需要做好 Tanh 的最终表现会比 ReLU 差的心理准备。”

我们曾反复说过，输入层的“激活函数” $\phi^{(0)}$ 其实是数据预处理对应的函数，输出层的“激活函数” $\phi^{(m)}$ 其实是变换函数。对于 $\phi^{(0)}$ ，一个较常见的定义是第 1 章提到过的标准化：

$$\phi_0(x) = \frac{x - \text{mean}}{\text{std}}$$

而对于 $\phi^{(m)}$ ，最常见的就是应用 Softmax 函数，它会将输出层得到的输入

$$\mathbf{h}^{(m)} = \mathbf{W}_0^{(m)} + \mathbf{W}^{(m)} \mathbf{o}^{(m-1)}$$

视为正比于概率向量的对数，并通过简单的变换来得到概率输出：

$$\phi^{(m)}(\mathbf{h}^{(m)}) = \left(\varphi^{(1)}(h^{(m)(1)}), \varphi^{(2)}(h^{(m)(2)}), \dots, \varphi^{(n^{(m)})}(h^{(m)(n^{(m)})}) \right)^T$$

其中

$$\mathbf{h}^{(m)} = \left(h^{(m)(1)}, h^{(m)(2)}, \dots, h^{(m)(n^{(m)})} \right)^T$$

且

$$\varphi^{(i)}(h^{(m)(i)}) \triangleq \frac{e^{h^{(m)(i)}}}{\sum_{j=1}^{n^{(m)}} e^{h^{(m)(j)}}}$$

由此不难看出

$$\sum_{i=1}^{n^{(m)}} \varphi^{(i)}(h^{(m)(i)}) = 1$$

从而 $\phi^{(m)}(\mathbf{h}^{(m)})$ 确实是一个概率输出。

在本节的最后，我们来简要介绍一下 SELU 函数以开阔视野。SELU 是在论文 *Self-Normalizing Neural Networks* 中被首次提出来的，该论文发表于 2017 年 9 月 7 日，正文只有 9 页，但附录中的数学证明则整整有 92 页，这直接导致有人戏称该论文的附录是“由 LSTM（一种常见的循环神经网络结构）生成的”。不过虽说可能直到现在也没人把所有证明都完整地看过一遍，但已经有不少人复现出论文中提到的效果。具体而言，论文声称，如果使用特殊的权值矩阵初始化方法的话，配合上 SELU 函数，就能使每个神经元的输出值都趋向于服从一个

均值为 0、方差为 1 的标准正态分布。

虽说这个性质极为漂亮，不少人也进行了复现并声称确实如此，但笔者个人感觉它还需要实际应用的考验。如果真的有落地的产品使用了 SELU 这个激活函数而且表现优异的话，那么它很有可能就是深度学习的未来。

3.2.4 损失函数

除了激活函数以外，神经网络中另外一个极其重要的元素就是损失函数了。如果说激活函数是神经网络非线性的来源，那么损失函数就是神经网络在拥有非线性能力后是否能够合理进行训练的关键。

我们在第 1 章中说过，损失函数是某个样本上的损失，而代价函数则是某个空间中损失函数的期望。除了计算方法上的区别以外，从应用的角度来看，可以认为损失函数是在做理论推导时用的函数，而代价函数则是在实际操作中用的函数。

由于本章偏向于介绍基础性的东西，所以我们在后续讨论中将统一使用损失函数（Loss Function）而不是代价函数（Cost Function）。

损失函数的直观意义非常明确，在前文也有所提及：它是模型在某个样本 (\mathbf{x}, \mathbf{y}) 上的表现。我们认为模型表现得越差就意味着损失越大，所以相应的损失函数的值就应该越大。回忆 2.3 节介绍过的梯度下降算法，我们需要对损失函数求梯度以更新模型的参数。那么一个比较自然的想法就是：我们应该在模型表现得比较差时，让参数更新得快一些，即梯度应该大一些。换句话说，应该将损失函数设计得有这样一个性质：在函数值比较大时，梯度也应该比较大。

注意：这里所说的“梯度也应该比较大”指的是“梯度的模长也应该比较大”或“梯度的各个维度上的数的绝对值也应该比较大”。

由于之前对梯度下降的应用还处于初级阶段（我们介绍的感知机和 SVM 的算法其实都只是非常朴素的算法），所以我们只关心了“表现越差则函数值越差”，却并没有太过关注“函数值越大则梯度应该越大”这个性质。而在神经网络中，梯度下降及其各种优化可谓是整个模型的根基，所以我们必须仔细地挑选，设计损失函数以使其拥有这个我们之前忽视掉的性质。相关的讨论我们会放在 3.3.2 节中进行，彼时读者将发现损失函数的选取在很大程度上会取决于输出层的变换函数。不过对于本节讨论的前向传导算法而言，暂时还不需要关心这些细节。

因此接下来我们就介绍几个常用的损失函数，希望通过对这些损失函数的说明，大家能够对损失函数的选择和构造生成自己的感悟和理解。在此之前，我们需要复习一遍符号约定。不失一般性，我们先以分类问题为例进行讨论：

- 假设样本为 (\mathbf{x}, \mathbf{y}) 。
- 假设共有 K 类： $\{c_1, \dots, c_K\}$ ，同时假设 \mathbf{y} 属于第 k 类。
- 假设计论的神经网络模型为 G ，其输出的（概率）向量为 $G(\mathbf{x}) = \mathbf{o}^{(m)}$ ，

其中 $\mathbf{x} \in \mathbb{R}^n$ 、 $\mathbf{y} \in \mathbb{R}^K$ ，且我们规定 \mathbf{y} 是 OneHot Encoding 后的标签：

$$\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^T, \quad \mathbf{y} = (y^{(1)}, y^{(2)}, \dots, y^{(K)})^T = (0, \dots, \overset{k}{1}, \dots, 0)^T$$

- （欧氏）距离损失函数（Mean Squared Error，常简称为 MSE）

$$L(G(\mathbf{x}), \mathbf{y}) = \|\mathbf{o}^{(m)} - \mathbf{y}\|^2 = \sum_{i=1}^k [o^{(m)(i)} - y^{(i)}]^2$$

该损失函数的直观意义是明确的：模型的输出值 $G(\mathbf{x})$ 和真值 \mathbf{y} 的（欧氏）距离越大，损失就会越大，反之就越小。需要指出的是，MSE 也常常被用在回归问题中，所以即使 $\mathbf{o}^{(m)}$ 不是概率向量也没有太大关系。

- 交叉熵（Cross Entropy）损失函数

$$\begin{aligned} L(G(\mathbf{x}), \mathbf{y}) &= -[\mathbf{y}^T \log \mathbf{o}^{(m)} + (1 - \mathbf{y})^T \log(1 - \mathbf{o}^{(m)})] \\ &= -\sum_{i=1}^K [y^{(i)} \log o^{(m)(i)} + (1 - y^{(i)}) \log(1 - o^{(m)(i)})] \end{aligned}$$

交叉熵的这种定义方式一般适用于二分类问题，此时有

$$\mathbf{y} = (y^{(1)}, y^{(2)})^T = (y^{(1)}, 1 - y^{(1)})^T$$

由于 \mathbf{y} 是一个概率向量，所以也有

$$\mathbf{o}^{(m)} = (o^{(m)(1)}, 1 - o^{(m)(2)})^T$$

从而

$$L(G(\mathbf{x}), \mathbf{y}) = -2[y^{(1)} \log o^{(m)(1)} + (1 - y^{(1)}) \log(1 - o^{(m)(1)})]$$

对于多分类问题，我们通常会采用下面马上就要介绍的 log-likelihood 损失函数。此外，对于一类比较特殊的问题——多类别（multi-label）问题，我们也常常会用交叉熵函数，感兴趣的读者可以参阅 <https://stats.stackexchange.com/questions/207794/what-loss-function-for-multi-class-multi-label-classification-tasks-in-neural-n>。

- log-likelihood 损失函数

$$L(G(\mathbf{x}), \mathbf{y}) = -\log o^{(m)(k)}$$

换句话说，log-likelihood 即为模型预测的第 k 个类别（真值 \mathbf{y} 所属的类别）的概率的负对数。需要指出的是，在多分类下使用 log-likelihood 虽然是很常用的手法，但是使用它是有一定的前提条件的，我们会在 3.3 节进行相关讨论。

3.3* 反向传播算法

这一节要讲的就是让人“闻风丧胆”的反向传播（BackPropagation，常简称为 BP）算法了。由于现在成熟的框架数不胜数，而且每个框架基本都有自动求梯度的功能（即自动 BP），所以本节内容只是作为附加内容进行介绍。能够理解整套 BP 算法自然是很棒的一件事，不过即使不能完全理解 BP，也不会对本书接下来讲解的重点内容带来麻烦。

3.3.1 算法概述

首先我们要明确反向传播算法的目的和原理。我们已经反复强调过，神经网络到目前为止还很依赖于梯度下降算法，而梯度下降算法要求我们算出各个参数的梯度。而由于神经网络模型较为复杂，所以直接对每个参数求梯度将会带来无法忍受的时间开销。反向传播算法（BP）正是一个能够高效计算各个参数梯度的方法，它能利用第 2 章介绍过的链式法则，一层一层地将网络中的梯度算出来（注意，神经网络的基本组成单元是层）。

注意：我们在第 2 章介绍链式法则时用了上标来代指维度，而由于本章中上标代指的是层数，所以我们会改用下标来代指维度。

下面就来正式地推导一下相应的公式，为此我们需要复习一下符号约定：

- 假设样本为 (\mathbf{x}, \mathbf{y}) 。
- 假设一共有 $m + 1$ 层，其中 $(i = 0, 1, \dots, m)$ 。
 - 每层的偏置量、权值矩阵分别为 $\mathbf{W}_0^{(i)}$ 和 $\mathbf{W}^{(i)}$ （注意 $\mathbf{W}_0^{(0)}$ 、 $\mathbf{W}^{(0)}$ 没有定义）。
 - 每层的输入和输出值分别为 $\mathbf{h}^{(i)}$ 和 $\mathbf{o}^{(i)}$ 。
 - 每层对应的激活函数为 $\phi^{(i)}$ 。
 - 每层有 n_i 个神经元。

那么模型的前向传导算法即为

$$\begin{aligned}\mathbf{h}^{(i)} &= \mathbf{W}_0^{(i)} + \mathbf{W}^{(i)} \mathbf{o}^{(i-1)} \\ \mathbf{o}^{(i)} &= \phi^{(i)}(\mathbf{h}^{(i)}) \\ G(\mathbf{x}) &= \mathbf{o}^{(m)} = \phi^{(m)}(\mathbf{h}^{(m)})\end{aligned}$$

接下来我们就看看 BP 的具体步骤。首先要根据损失函数求出输出层对应的梯度，由于样本和损失函数都已经给定，所以第一个已知量是损失函数对模型输出的梯度：

$$\nabla_{\mathbf{o}^{(m)}} L(\mathbf{o}^{(m)}, \mathbf{y})$$

为了简洁，把它简记为 $\nabla_{\mathbf{o}^{(m)}} L$ 。我们要做的，就是从这唯一的一个已知量出发，将所有参数的梯度逐层地算回来。首先由链式法则可以直接得出

$$\nabla_{\mathbf{h}^{(m)}} L = \sum_{i=1}^K \frac{\partial L}{\partial \mathbf{o}^{(m)(i)}} \cdot \nabla_{\mathbf{h}^{(m)}} \mathbf{o}^{(m)(i)}$$

为了简洁，我们记 $\delta^{(i)} \triangleq \nabla_{\mathbf{h}^{(i)}} L$ ($i = 1, 2, \dots, m$)，那么就有

$$\delta^{(m)} = \sum_{i=1}^K \frac{\partial L}{\partial o^{(m)(i)}} \cdot \nabla_{\mathbf{h}^{(m)}} o^{(m)(i)}$$

至此，输出层的相关梯度就计算完毕了。下面来看一下剩下的层结构中相关梯度的计算。具体而言，对 $i = 0, 1, \dots, m-1$ ，都有

$$\mathbf{h}^{(i+1)} = \mathbf{W}_0^{(i+1)} + \mathbf{W}^{(i+1)} \mathbf{o}^{(i)}$$

从而有（注意， $\mathbf{h}^{(i+1)}$ 是一个 $n^{(i+1)}$ 维的列向量）

$$\nabla_{\mathbf{W}_0^{(i+1)}} L = \sum_{j=1}^{n^{(i+1)}} \frac{\partial L}{\partial h_j^{(i+1)}} \cdot \nabla_{\mathbf{W}_0^{(i+1)}} h_j^{(i+1)}$$

不难看出， $\nabla_{\mathbf{W}_0^{(i+1)}} h_j^{(i+1)}$ 就是一个 OneHot 向量：

$$\nabla_{\mathbf{W}_0^{(i+1)}} h_j^{(i+1)} = \begin{pmatrix} 0, \dots, 1, \dots, 0 \end{pmatrix}^T$$

而由于是从后往前反向传播梯度的，而且我们已经得到了 $\delta^{(m)}$ ，并能够利用 $\delta^{(i+1)}$ 来计算出 $\delta^{(i)}$ （具体公式与相应推导马上会在后文给出），所以从 $i = m-1$ 开始，都可以认为我们已知 $\delta^{(i+1)}$ 。注意

$$\delta^{(i+1)} = \left(\frac{\partial L}{\partial h_1^{(i+1)}}, \frac{\partial L}{\partial h_2^{(i+1)}}, \dots, \frac{\partial L}{\partial h_{n^{(i+1)}}^{(i+1)}} \right)^T$$

从而结合 $\nabla_{\mathbf{W}_0^{(i+1)}} h_j^{(i+1)}$ 的形式就能直接得出

$$\nabla_{\mathbf{W}_0^{(i+1)}} L = \delta^{(i+1)}$$

这就是偏置量的梯度的计算公式，可见还是很简洁的。权值矩阵梯度的计算会稍微复杂一些，不过道理都是相通的。首先我们有

$$\nabla_{\mathbf{W}^{(i+1)}} L = \sum_{j=1}^{n^{(i+1)}} \frac{\partial L}{\partial h_j^{(i+1)}} \cdot \nabla_{\mathbf{W}^{(i+1)}} h_j^{(i+1)}$$

其中，不难看出 $\nabla_{\mathbf{W}^{(i+1)}} h_j^{(i+1)}$ 是一个除了第 j 行是 $\mathbf{o}^{(i)}$ 的转置 ($\mathbf{o}^{(i)T}$)、其余元素都是 0 的矩阵，从而

$$\nabla_{\mathbf{W}^{(i+1)}} L = \delta^{(i+1)} \times \mathbf{o}^{(i)T}$$

这就是权值矩阵的梯度的计算公式。所以现在我们唯一还需要做的，就是根据 $\delta^{(i+1)}$ 来计算出 $\delta^{(i)}$ 。注意

$$\nabla_{\mathbf{o}^{(i)}} L = \sum_{j=1}^{n^{(i+1)}} \frac{\partial L}{\partial h_j^{(i+1)}} \cdot \nabla_{\mathbf{o}^{(i)}} h_j^{(i+1)}$$

其中，不难看出 $\nabla_{\mathbf{o}^{(i)}} h_j^{(i+1)}$ 就是 $\mathbf{W}^{(i+1)}$ 的第 j 行，从而

$$\nabla_{\mathbf{o}^{(i)}} L = \mathbf{W}^{(i+1)\top} \boldsymbol{\delta}^{(i+1)}$$

又由 $\mathbf{o}^{(i)} = \phi^{(i)}(\mathbf{h}^{(i)})$ 、激活函数 $\phi^{(i)}$ 一般而言是 element wise 函数，以及自变量的梯度公式，不难得知

$$\boldsymbol{\delta}^{(i)} \triangleq \nabla_{\mathbf{h}^{(i)}} L = \nabla_{\mathbf{o}^{(i)}} L \odot \phi'^{(i)}(\mathbf{h}^{(i)})$$

从而就有

$$\boldsymbol{\delta}^{(i)} = \mathbf{W}^{(i+1)\top} \boldsymbol{\delta}^{(i+1)} \odot \phi'^{(i)}(\mathbf{h}^{(i)})$$

即我们确实能利用 $\boldsymbol{\delta}^{(i+1)}$ 和已知的参数来推导出 $\boldsymbol{\delta}^{(i)}$ ，于是就完成了所有 BP 算法的公式推导。

不过请注意，我们的出发点—— $\boldsymbol{\delta}^{(m)} = \sum_{i=1}^K \frac{\partial L}{\partial \mathbf{o}^{(m)(i)}} \cdot \nabla_{\mathbf{h}^{(m)}} \mathbf{o}^{(m)(i)}$ 其实还并没有把具体的数值算出来，我们将会在下节中较为详细地讨论这两者的计算方式。

3.3.2 损失函数的选择

在 3.2.4 节的开头我们曾经说过，损失函数应该拥有“函数值越大则相应参数的梯度越大”的性质。而从 BP 算法的起源—— $\boldsymbol{\delta}^{(m)} = \sum_{i=1}^K \frac{\partial L}{\partial \mathbf{o}^{(m)(i)}} \cdot \nabla_{\mathbf{h}^{(m)}} \mathbf{o}^{(m)(i)}$ 来看，为了满足这个性质，我们需要把损失函数的梯度本身（ $\nabla_{\mathbf{o}^{(m)}} L = \left(\frac{\partial L}{\partial \mathbf{o}^{(m)(1)}}, \dots, \frac{\partial L}{\partial \mathbf{o}^{(m)(K)}} \right)^\top$ ）以及变换函数直接求出的梯度值（ $\nabla_{\mathbf{h}^{(m)}} \mathbf{o}^{(m)(i)}$ ）放在一起考虑，这也是为什么在 3.2.4 节中我们说损失函数 L 的选择通常取决于输出层的变换函数 $\phi^{(m)}$ 。换句话说，我们常常选择的是一个变换函数和损失函数的组合，而不是一个单纯的损失函数。

为了能够选出合适的组合，我们需要先知道各个损失函数的梯度及变换函数的直接梯度分别是什么样的。我们下面会先列举一下 3.2.4 节中出现过的损失函数的梯度和两种常见的变换函数——Sigmoid 和 Softmax 的导数，然后通过一些分析来表明哪些组合是不能放在一起的，哪些组合又是会被经常使用的。

首先来看各个损失函数的梯度。假设我们现在做的是 K 分类问题，那么公式中的各个变量都是 K 维向量。

- MSE 损失函数

$$L_{MSE}(G(\mathbf{x}), \mathbf{y}) = \|\mathbf{o}^{(m)} - \mathbf{y}\|^2 \Rightarrow \nabla_{\mathbf{o}^{(m)}} L_{MSE} = 2[\mathbf{o}^{(m)} - \mathbf{y}]$$

- Cross Entropy 损失函数

$$L_{\text{CE}}(G(\mathbf{x}), \mathbf{y}) = -[\mathbf{y}^T \log \mathbf{o}^{(m)} + (1 - \mathbf{y})^T \log(1 - \mathbf{o}^{(m)})]$$

为了简洁, 对于两个 K 维向量 \mathbf{y} 和 $\mathbf{o}^{(m)}$, 我们记

$$\frac{\mathbf{y}}{\mathbf{o}^{(m)}} \triangleq \left(\frac{y^{(1)}}{o^{(m)(1)}}, \frac{y^{(2)}}{o^{(m)(2)}}, \dots, \frac{y^{(K)}}{o^{(m)(K)}} \right)^T$$

那么就有

$$\nabla_{\mathbf{o}^{(m)}} L_{\text{CE}} = - \left[\frac{\mathbf{y}}{\mathbf{o}^{(m)}} - \frac{1 - \mathbf{y}}{1 - \mathbf{o}^{(m)}} \right] = \frac{\mathbf{o}^{(m)} - \mathbf{y}}{\mathbf{o}^{(m)}(1 - \mathbf{o}^{(m)})}$$

- log-likelihood 损失函数

$$L_{\log}(G(\mathbf{x}), \mathbf{y}) = -\log o^{(m)(k)} \Rightarrow \nabla_{\mathbf{o}^{(m)}} L_{\log} = \left(0, \dots, -\frac{1}{o^{(m)(k)}}, \dots, 0 \right)^T$$

以上就是三种常见损失函数的梯度, 下面我们来看看两种常见变换函数的直接梯度。

- Sigmoid 函数

$$o^{(m)(i)} = \sigma(h^{(m)(i)})$$

其中

$$\sigma(x) \triangleq \frac{1}{1 + e^{-x}}$$

注意到

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \sigma(x)[1 - \sigma(x)]$$

从而

$$\nabla_{\mathbf{h}^{(m)}} o^{(m)(i)} = \left(0, \dots, o^{(m)(i)}[1 - o^{(m)(i)}], \dots, 0 \right)^T$$

注意到 Sigmoid 变换函数其实是 element wise 函数, 所以 $\delta^{(m)}$ 此时可以改写成

$$\delta^{(m)} = \nabla_{\mathbf{o}^{(m)}} L \odot \phi'^{(m)}(\mathbf{h}^{(m)})$$

其中

$$\phi'^{(m)}(\mathbf{h}^{(m)}) = \mathbf{o}^{(m)}[1 - \mathbf{o}^{(m)}]$$

- Softmax 函数

$$o^{(m)(i)} = \varphi^{(i)}(h^{(m)(i)})$$

其中

$$\varphi^{(i)}(h^{(m)(i)}) \triangleq \frac{e^{h^{(m)(i)}}}{\sum_{j=1}^K e^{h^{(m)(j)}}}$$

注意，由于 $\varphi^{(i)}$ 会受所有的 K 个 $h^{(m)(i)}$ 影响，所以求导时也要求 K 遍，而且要分情况讨论。为了简洁，我们用 $e^{(i)}$ 表示 $e^{h^{(m)(i)}}$ ，从而

$$\frac{\partial \varphi^{(i)}}{\partial h^{(m)(j)}} = \begin{cases} -\frac{e^{(i)}e^{(j)}}{(\sum_{j=1}^K e^{(j)})^2} & , \text{ if } i \neq j \\ \frac{e^{(i)} \sum_{j=1}^K e^{(j)} - e^{(i)}e^{(i)}}{(\sum_{j=1}^K e^{(j)})^2} & , \text{ if } i = j \end{cases} = \begin{cases} -\varphi^{(i)}\varphi^{(j)} & , \text{ if } i \neq j \\ \varphi^{(i)}(1 - \varphi^{(i)}) & , \text{ if } i = j \end{cases}$$

注意， $o^{(m)(i)} = \varphi^{(i)}$ ，从而就有

$$\nabla_{\mathbf{h}^{(m)}} o^{(m)(i)} = \left(-o^{(m)(i)}o^{(m)(1)}, \dots, o^{(m)(i)}[1 - o^{(m)(i)}], \dots, -o^{(m)(i)}o^{(m)(K)} \right)^T$$

以上就是两种常见变换函数的直接梯度，下面我们要做的，就是发掘损失函数梯度及变换函数梯度的性质，从而挑选出合适的组合了。

首先来看 MSE 损失函数。MSE 既能用在回归问题中也能用在分类问题中，可谓是一个“万金油”的损失函数。不过也正因此，不难想象它在分类问题时的表现将会比专门针对分类问题设计的损失函数（比如说 Cross Entropy 和 log-likelihood）要差一些。事实上，从它的梯度的形式 $\nabla_{\mathbf{o}^{(m)}} L_{\text{MSE}} = 2[\mathbf{o}^{(m)} - \mathbf{y}]$ 可以看出，当问题是分类问题时，这个梯度是一个有界的 K 维向量，这是因为 $\mathbf{o}^{(m)}$ 、 \mathbf{y} 各个维度上的取值在分类问题下通常都在 0 到 1 之间。注意我们前面在介绍 Sigmoid 函数时曾经提过，它的两端非常平坦，从而相应的梯度就会趋近于 0。而 $\nabla_{\mathbf{o}^{(m)}} L_{\text{MSE}}$ 却是有界的，这就会导致在 $\phi^{(m)}$ 为 Sigmoid 时， $\delta^{(m)} = \nabla_{\mathbf{o}^{(m)}} L \odot \phi'^{(m)}(\mathbf{h}^{(m)})$ 也趋于 0。具体而言，即使某个样本中标签 \mathbf{y} 为 $(0,1)^T$ ，但模型的输出 $\mathbf{o}^{(m)}$ 却趋于 $(1,0)^T$ ，即模型的表现不能再差、损失函数值不能再大，也会由于 $\phi^{(m)}$ 是 Sigmoid 函数而导致：

$$\phi^{(m)}(\mathbf{h}^{(m)}) = \mathbf{o}^{(m)} \rightarrow (1,0)^T \Rightarrow \phi'^{(m)}(\mathbf{h}^{(m)}) = \mathbf{o}^{(m)}[1 - \mathbf{o}^{(m)}] \rightarrow (0,0)^T$$

同时由于 $\nabla_{\mathbf{o}^{(m)}} L_{\text{MSE}}$ 有界，就会有

$$\delta^{(m)} \rightarrow (0,0)^T$$

这就严重违反了“损失越大则相应梯度越大”的性质，所以 MSE 损失函数是绝对不能和 Sigmoid 函数组合在一起使用的。

由上述的推导可以看出，之所以 MSE 和 Sigmoid 之间的配合较差，是因为 Sigmoid 求梯度之后是 $\mathbf{o}^{(m)}[1 - \mathbf{o}^{(m)}]$ ，在 $\mathbf{o}^{(m)}$ 趋于 0 和 1 时会迅速衰减，而 MSE 的梯度无法阻止这种衰减（因为其梯度是有界的）。所以如果要选择和 Sigmoid 函数搭配的损失函数，要找一个能够阻止 $\mathbf{o}^{(m)}[1 - \mathbf{o}^{(m)}]$ 衰减的东西。说到这里，可能已经有不少读者发现了：在 Cross Entropy 损失函数的梯度中，分母不就是 $\mathbf{o}^{(m)}[1 - \mathbf{o}^{(m)}]$ 吗？事实上，Sigmoid 搭配 Cross Entropy 确实是一种常见的做法，此时不难看出

$$\delta^{(m)} = o^{(m)}[1 - o^{(m)}] \odot \frac{o^{(m)} - y}{o^{(m)}(1 - o^{(m)})} = o^{(m)} - y$$

这就非常完美地符合了“损失越大则梯度越大”的性质，因为损失越大意味着 $o^{(m)}$ 和 y 差得越远，从而 $\delta^{(m)}$ 也就会越大。

不过前文曾经说过，我们一般会要求 $o^{(m)}$ 是一个概率输出。如果用 Sigmoid 作为变换函数的话，我们就只能做到 $o^{(m)}$ 的每个维度都是 0 到 1 之间的数，而不能保证（或说不可能做到） $o^{(m)}$ 本身是一个概率输出。这里需要进行补充说明：到目前为止，我们讨论过的分类问题中，正确的类别都有且仅有一个。换句话说，真实的标签 y 是一个 OneHot 向量。而 OneHot 向量其实是比较极端的概率向量，所以我们用一个概率输出去拟合它是合理的。但在现实生活中，其实有这样一类问题，每个特征向量所属的正确类别可能不止一个（即前文提到过的多类别问题）。比如我给你一张图片，让你输出图片中是否有人，是否有猫，是否有狗，那么此时就会有很多种可能的情况，比如人、猫、狗都没有，那么对应的标签就是 $(0,0,0)^T$ ；比如既有人又有猫，那么对应的标签就是 $(1,1,0)^T$ ，依此类推。此时，再用概率输出去拟合显然就不合理了。相反的，我们应该只要求输出的每个维度都在 0 到 1 之间，此时用 Sigmoid 做变换函数就是一种非常常见的选择。

也正因此，专门用于多分类的 log-likelihood 损失函数从直观上就不应该和 Sigmoid 一起用，而应该和 Softmax 一起用，因为 log-likelihood 和 Softmax 都认为正确的类别有且仅有一个，而 Sigmoid 则一般用于正确类别个数不定的情况，所以这两者应该是不兼容的。事实上从理论上来看也确实如此，下面就来推导一下相应的公式。

假设当前正确的类别为 $k_1 \sim k_q$ ，那么按照 log-likelihood 的直观解释，可以强行定义一种新的 log-likelihood 损失函数：

$$L(G(\mathbf{x}), \mathbf{y}) = - \sum_{i=1}^q \log o^{(m)(k_i)}$$

从而

$$\nabla_{o^{(m)}} L = \left(0, \dots, -\frac{1}{o^{(m)(k_1)}}, \dots, -\frac{1}{o^{(m)(k_q)}}, \dots, 0 \right)^T$$

可以看到它是一个很稀疏的向量。注意，Sigmoid 的导数为

$$\phi'^{(m)}(\mathbf{h}^{(m)}) = o^{(m)}[1 - o^{(m)}]$$

从而不难看出

$$\delta^{(m)} = \left(0, \dots, o^{(m)(k_1)} - 1, \dots, o^{(m)(k_q)} - 1, \dots, 0 \right)^T$$

即得到的梯度也是非常稀疏的。在极端情况下，如果当前样本中没有正确类别（即 $q = 0$ ）

的话，那么直接就有

$$\delta^{(m)} = (0, 0, \dots, 0)^T$$

从而梯度直接就会变为 0 向量。这种稀疏性将导致模型的训练极其缓慢且不稳定，所以确实从理论上来说也不应该使用 Sigmoid+log-likelihood 的组合。

注意：从上述讨论中还可以看出，如果我们没有使用变换函数（或说变换函数是恒同映射）的话，也不应该使用 log-likelihood，因为得到的梯度同样将会是非常稀疏的。

而对于 Softmax 函数来说，由于

$$\frac{\partial \varphi^{(i)}}{\partial h^{(m)(j)}} = \begin{cases} -\varphi^{(i)}\varphi^{(j)} & , \text{ if } i \neq j \\ \varphi^{(i)}(1 - \varphi^{(i)}) & , \text{ if } i = j \end{cases}$$

从而

$$\begin{aligned} \delta^{(m)} &= \sum_{i=1}^K \frac{\partial L}{\partial o^{(m)(i)}} \cdot \nabla_{h^{(m)}} o^{(m)(i)} = -\frac{1}{o^{(m)(k)}} \cdot \nabla_{h^{(m)}} o^{(m)(k)} \\ &= -\frac{1}{o^{(m)(k)}} \cdot \left(-o^{(m)(k)} o^{(m)(1)}, \dots, o^{(m)(k)} [1 - o^{(m)(k)}], \dots, -o^{(m)(k)} o^{(m)(K)} \right)^T \\ &= \left(o^{(m)(1)}, \dots, o^{(m)(k)} - 1, \dots, o^{(m)(K)} \right)^T \end{aligned}$$

注意到

$$\mathbf{y} = \left(0, \dots, 1, \dots, 0 \right)^T$$

从而

$$\delta^{(m)} = \mathbf{o}^{(m)} - \mathbf{y}$$

可以看到，这和 Sigmoid+Cross Entropy 的公式一模一样！所谓数学之乐趣，概莫如是。

至此，我们就把如何得到各个参数的梯度的方法大致介绍了一遍，接下来我们将介绍如何利用这些梯度来更新相应的参数。

3.4 参数的更新

我们在第 2 章介绍 SVM 时曾描述过如何简单地利用梯度下降来更新参数，这种朴素的梯度下降法通常被称为 Vanilla Update。此外，当时我们曾经说过两个优化的方向：

- 不是简单地把负梯度作为更新方向，而是利用更多的（历史）属性来定出更新方向。
- 不单纯地把学习速率设成常量，而设法让其能够“适应算法”并做出调整。

其中，我们通常会把第二个优化方向涵盖在第一个优化方向中，因为学习速率的调整通常可以反映在更新方向的调整上（只需把调整的比率乘在优化后的更新方向上）。为此，本节我们只讨论如何优化更新方向。我们先给出相应的公式，然后给出这种公式的来由及它比原始梯度下降法要更好的直观解释。

为了方便讨论，我们统一使用 \mathbf{w} 代指需要更新的参数，用 $\nabla^{(t)}\mathbf{w}$ 和 $\Delta^{*(t)}\mathbf{w}$ 代指第 t 步迭代中得到的梯度和优化后的更新方向，用 η 代指学习速率。

3.4.1 Vanilla Update

Vanilla Update 就是我们在第 2 章介绍的朴素梯度下降法。不过虽说它名字叫“朴素”，但在实际应用中却少不了它。一方面我们可以拿它来做基准线，即定出一个下界，另一方面在时间充足的情况下，Vanilla Update 最后反而往往有可能比后文中我们介绍的种种优化算法的表现要好。这是因为虽然它的收敛速度慢，但是它的收敛相对来说也会更稳一些。

而它对应的更新方向想必各位读者已经知道了——就是负梯度方向

$$\Delta_V^{*(t)}\mathbf{w} = -\eta\nabla^{(t)}\mathbf{w}$$

3.4.2 Momentum Update

动量（Momentum）法的更新方向计算公式为

$$\begin{aligned}\Delta_M^{*(t)}\mathbf{w} &= \rho\Delta_M^{*(t-1)}\mathbf{w} - \eta\nabla^{(t)}\mathbf{w}_t \\ &= \rho\Delta_M^{*(t-1)}\mathbf{w} + \Delta_V^{*(t)}\mathbf{w}\end{aligned}$$

它的直观意义比较明确：想要模仿物理上的“惯性”。具体而言，想象一下我们的损失函数是一个山谷，由于我们的目的是让损失函数达到最小，所以可以想象我们的目的是达到谷底。假设现在我们算出了梯度 $\nabla^{(t)}\mathbf{w}_t$ ，那么就可以把 $\nabla^{(t)}\mathbf{w}_t$ 想象成当前山谷的坡度，并把学习速率 η 想象成沿坡度行走的速度。

对于 Vanilla Update 而言，常常会出现上一秒还在猛地往前冲，这一秒却要突然拼命向后撤的现象，这种“反物理”的行进模式正是 Momentum Update 所被诟病并想要解决的。这种违反物理直观的模型的根本原因正在于它没有惯性，所以 Momentum Update 就引入了类似惯性的东西，从而就有了本节开头的那个公式。其中，各个变量的物理意义如下所示。

- 坡度 $\nabla^{(t)}\mathbf{w}_t$ ：当前的“动力”。
- $\Delta_M^{*(t-1)}\mathbf{w}$ ：前 $t-1$ 步累积下来的“速度”。
- ρ ：惯性的大小，描述了过去速度的影响力。不难看出，当 $\rho = 0$ 时，Momentum Update 就是 Vanilla Update，所以说动量法是包容朴素梯度下降的。

一般来说，我们不会把 ρ 设置为一个常量，而会把它设置成一个会随训练过程的推进而变动的变量。一种常见的做法是将 ρ 的初始值设为 0.5 并逐步将它加大至 0.99，这是因为训练刚开始时的梯度会比较大而训练后期梯度会变小，通过逐步调大 ρ ，我们能够使更新的步伐一直保持在

比较大的水平。

当然也不是说只能用这种方法来调整 ρ 的值，对于一些特殊的情况，确实有更好且更具针对性的更新策略。

3.4.3 Nesterov Momentum Update

从名字上不难想象，Nesterov Momentum 和 Momentum 有千丝万缕的关系。它由 Ilya Sutskever 在 Nesterov 相关工作（Nesterov Accelerated Gradient，常简称为 NAG。为了简洁，我们下文也统一用 NAG 来代指 Nesterov Momentum）的启发下提出，在凸优化问题下的收敛性会比传统的 Momentum 好。其更新公式如下：

$$\Delta_N^{*(t)} \mathbf{w} = -\rho \Delta_N^{*(t-1)} \mathbf{w} + (1 + \rho) \cdot (\rho \Delta_N^{*(t-1)} \mathbf{w} - \eta \nabla^{(t)} \mathbf{w}_t)$$

这个“原始”公式可能会让人看得头昏脑胀，不过如果稍做变形的话，就会发现它和传统的 Momentum 相差不多：

$$\begin{aligned} \Delta_N^{*(t)} \mathbf{w} &= -\rho \Delta_N^{*(t-1)} \mathbf{w} + (\rho + \rho^2) \Delta_N^{*(t-1)} \mathbf{w} - (1 + \rho) \eta \nabla^{(t)} \mathbf{w}_t \\ &= \rho^2 \Delta_N^{*(t-1)} \mathbf{w} - \eta \nabla^{(t)} \mathbf{w}_t - \rho \eta \nabla^{(t)} \mathbf{w}_t \\ &= \rho (\rho \Delta_N^{*(t-1)} \mathbf{w} - \eta \nabla^{(t)} \mathbf{w}_t) - \eta \nabla^{(t)} \mathbf{w}_t \\ &= \rho \Delta_M^{*(t)} \mathbf{w} + \Delta_V^{*(t)} \mathbf{w} \end{aligned}$$

从直观上来看，该公式意味着 NAG 是把 Momentum 得到的更新方向当作惯性来源的。换句话说，NAG 让算法有了某种“前瞻性”——它不是简单地用当前的速度作为惯性的来源，而是用 Momentum 得到的“带有惯性的速度”作为惯性的来源。

3.4.4 AdaGrad

无论 Momentum 还是 NAG，它们都是利用历史信息来得到一个更好的更新方向。而我们在本节的开头曾经说过，还可以在优化算法内部加入调整学习速率的逻辑，而这正是 AdaGrad 及后面会介绍的 RMSProp 方法的思想。它们通过利用历史信息来调整学习速率，从而期望学习速率能够自适应于当前的训练状态。对于 AdaGrad 而言，其对应的公式为

$$\begin{aligned} \tilde{\nabla}^2 &\leftarrow \tilde{\nabla}^2 + \nabla^{(t)^2} \mathbf{w}_t \\ \Delta_{AG}^{*(t)} \mathbf{w} &= -\eta \cdot \frac{\nabla^{(t)} \mathbf{w}_t}{\tilde{\nabla} + \epsilon} \end{aligned}$$

需要注意的是，这里面的“累积梯度” $\tilde{\nabla}^2$ 和梯度 $\nabla^{(t)} \mathbf{w}_t$ 的形状是一致的。换句话说，第一个公式和第二个公式中的加法和乘法都是逐元素（element wise）操作的。

AdaGrad 的思想很直观：如果某个位置累积下来的梯度足够多的话，就认为这个位置上的训练已经进行得差不多了，所以就让对应梯度的影响变小。不过这种做法的缺陷是显而易见的：梯度的累积量 $\tilde{\nabla}^2$ 是单调上升的，它只能变大、不能变小。换句话说，我们只能将学习速率往小调，并不能把它调大，这当然是不尽合理的。极端的例子就是，如果一开始来了几个很大的梯

度值，那么就会直接“杀死”整个训练过程，因为后面得到的更新量都会很小。

3.4.5 RMSProp

RMSProp 对 AdaGrad 中累积梯度的单调性做出了改进，其对应的公式为

$$\begin{aligned}\tilde{\nabla}^2 &\leftarrow \rho \tilde{\nabla}^2 + (1 - \rho) \nabla^{(t)^2} \mathbf{w}_t \\ \Delta_R^{*(t)} \mathbf{w} &= -\eta \cdot \frac{\nabla^{(t)} \mathbf{w}_t}{\sqrt{\tilde{\nabla}^2} + \epsilon}\end{aligned}$$

可以看到它引入了衰减系数 ρ ，这就使得如果某段时间梯度一直很小，那么由于 ρ 的存在，累积梯度 $\tilde{\nabla}^2$ 就会以指数级的速度衰减，从而当前梯度的影响会以指数级的速度增强。换句话说，在 RMSProp 算法中，我们不仅能让当前的更新在一段剧烈的更新后“降温”，也能在一段平稳的更新后“升温”。

3.4.6 Adam

目前介绍过的这些优化算法都各有各的优点，而 Adam 算法算是这些算法的“集大成者”：它既能像 Momentum 系列算法那样优化更新方向，也能像 AdaGrad 系列算法那样调整学习速率。一般来说，如果不知道该选哪种优化算法的话，使用 Adam 常常是一个不错的选择。它的数学理论背景是相当复杂的，这里就只写出它的一个简化版的优化公式：

$$\begin{aligned}\tilde{\nabla}_1 &\leftarrow \beta_1 \tilde{\nabla}_1 + (1 - \beta_1) \nabla^{(t)} \mathbf{w}_t \\ \tilde{\nabla}_2^2 &\leftarrow \beta_2 \tilde{\nabla}_2^2 + (1 - \beta_2) \nabla^{(t)^2} \mathbf{w}_t \\ \Delta_A^{*(t)} \mathbf{w} &= -\eta \cdot \frac{\tilde{\nabla}_1}{\sqrt{\tilde{\nabla}_2^2} + \epsilon}\end{aligned}$$

从公式中可以看出，Adam 算法的分子就是由 Momentum 算法得到的、惯性为 β_1 的更新方向，而分母就是由 RMSProp 得到的、衰减系数为 β_2 的调整项。

3.5 TensorFlow 模型的基本框架

在进行神经网络的具体实现之前，熟悉一下 TensorFlow 本身是有必要的。本节将会介绍如何把 TensorFlow 模型的基本框架搭建起来，在这个框架下，我们不仅能够极为方便地实现出一个朴素的神经网络(50 行以内)，而且能够快速搭建诸如 CNN、RNN 等神经网络模型的变体。更一般地说，只要是能够使用梯度下降法来进行训练的模型，在此框架下都能进行快速实现。

在展示具体的代码之前，我们需要先知道 TensorFlow 的一些特点，然后再有针对性地进行相应的实现。不过由于 TensorFlow 的内容太多，所以我们只关注于核心的部分，对于其他一些比较细的知识点则不会展开叙述。如果确实想对它比较有完整的认知，可以参见它的官网教程：https://www.tensorflow.org/get_started/get_started 系列和 <https://www.tensorflow.org/tutorials/mandelbrot> 系列。

3.5.1 TensorFlow 的组成单元与基本思想

在一个大的层面上，我们可以这样去理解 TensorFlow：

- TensorFlow 的核心在于它能构建出一张“运算图（Graph）”，我们需要做的是往这张图里加入元素。
- TensorFlow 构建的运算图是静态图，搭建好之后无法动态地对其进行更改，但我们可以利用一些方法使得它在静态图内部拥有一定的动态性（比如利用 `tf.cond` 使模型可支持条件语句）。同时，通过利用“占位符（placeholder）”让 TensorFlow 拥有较强的灵活性。
- TensorFlow 基本的元素有如下三种：常量（Constant）、可训练的变量（Variable）和不可训练的变量（Variable(trainable=False)），而这三种基本元素又可以统称为张量（Tensor）。值得一提的是，一个机器学习模型中的参数，在 TensorFlow 中往往就是一个可训练的变量。
- TensorFlow 模型的训练通常分三步走：
 - 把算法的损失函数在运算图中表示出来。
 - 根据超参数设置来定义一个优化器。
 - 利用优化器和运算图中的损失函数来得到一个更新步骤（train_step），然后再通过执行 `sess.run(train_step)` 之类的语句来完成模型的迭代。

在本节中，我们会先把重点放在 TensorFlow 最基本的用法上，对于 placeholder 和模型的具体训练方式，我们会在具体介绍 TensorFlow 模型基本框架时进行说明。

注意：本书所用的 TensorFlow 版本为 1.4，而由于 TensorFlow 版本之间的兼容性较差，所以如果大家发现有些代码报错的话，很有可能是因为版本不对，届时只需根据更新文档做出微调即可。

首先来看看如何定义常量、可训练的变量、不可训练的变量，以及如何把它们之间的四则运算加入运算图 Graph 中。我们之前曾说过，TensorFlow 中的这三种基本元素可以统称为 Tensor，于是 TensorFlow 这个词本身就可以理解为“Tensor（在 Graph 中）的流动”。

```
01 # 导入 TensorFlow 库以进行后续操作
02 import tensorflow as tf
03
04 # 定义常量，同时把数据类型定义为能够进行 GPU 计算的 tf.float32 类型
05 x = tf.constant(1, dtype=tf.float32)
06 # 定义可训练的变量
07 y = tf.Variable(2, dtype=tf.float32)
08 # 定义不可训练的变量
09 z = tf.Variable(3, dtype=tf.float32, trainable=False)
10 x_add_y = x + y
11 y_sub_z = y - z
12 x_times_z = x * z
13 z_div_x = z / x
```

可以看到定义起来都很直观，而且单纯地将运算加入运算图其实并不需要调用什么函数，只需直接写出来即可。不过需要注意的是，如果想要更优雅地使用 TensorFlow 的话，我们就应该对一些关键的运算命名。比如在上面这个例子中，若 $x + y$ 是一个特别重要的运算的话，就应该把第 10 行改写为：

```
10 x_add_y = tf.add(x, y, name=name)
```

其中 `name` 是该运算对应的名字。许多人在初学 TensorFlow 时常常为了方便而不对关键步骤命名（笔者曾经也如此），这样会至少导致这样两个问题：

- 调试时无法快速定位到关键步骤。
- 使用 Tensorboard 可视化时无法将关键步骤区分出来。

所以从一开始就养成命名的习惯是比较好的选择。也正因此，在后文介绍神经网络的实现时，我们会把基本运算单元中的线性映射 $\mathbf{h}^{(k)} = \mathbf{W}_0^{(k)} + \mathbf{W}^{(k)}\mathbf{o}^{(k-1)}$ 写成类似于：

```
h = tf.add(tf.matmul(o, w), b, name="Linear_Output")
```

的形式，其中 `matmul` 是 TensorFlow 中矩阵乘法所对应的函数。

除了四则运算以外，TensorFlow 基本支持所有 `numpy` 中的方法，它们也都有 `name` 这个参数留给我们以方便进行命名，不过这些方法的名字和 `numpy` 中对应方法的名字会稍微有些不一样。以“求和”操作为例：

```
14 # 用 numpy 数组进行 Tensor 的初始化
15 x = tf.constant(np.array([[1, 2], [3, 4]]))
16 # TensorFlow 中对应于 np.sum 的方法
17 axis0 = tf.reduce_sum(x, axis=0)      # 将会得到值为 [ 4 6 ] 的 Tensor
18 axis1 = tf.reduce_sum(x, axis=1)      # 将会得到值为 [ 3 7 ] 的 Tensor
```

更多的操作方法可以参见 https://www.tensorflow.org/api_guides/python/math_ops。总之，通过四则运算、矩阵乘法以及这些丰富多彩的操作方法，我们就能把整个模型的所有运算步骤都写入运算图 `Graph` 中。

不过这就会很自然地想到这样一个问题：我们现在确实能把所有运算都写入 `Graph` 中了，可是究竟应该怎样提取出具体的数值呢？这就需要一个叫 `Session` 的东西了，而 `Graph` 和 `Session` 之间的关系可以这样来理解。

- `Graph` 中定义的是一套“运算规则”。
- `Session` 可以“启动”这一套由 `Graph` 定义的运算规则，而在启动的过程中，`Session` 可能会额外做三件事：
 - 赋予“运算规则”中一些“占位符”以具体的值，从而使得后续的运算能够进行。
 - 根据可能存在的占位符的取值和相应的运算规则，提取出想要的中间结果。
 - 如果启动的运算规则包括“更新参数（`train_step`）”这个运算规则的话，就更新相应的可训练的变量。

其中，“更新参数”和“占位符”的相关讨论会放在具体介绍模型基本框架时进行，这里我们就只说明“提取中间结果”是什么意思。比如现在 Graph 中有这么一套运算规则：

$$x = 1, \quad y = x + 1, \quad z = y + 1$$

而我只想要运算规则被启动之后 y 的运算结果，那么相应的代码实现将如下：

```
01 x = tf.constant(1)
02 y = x + 1
03 z = y + 1
04 print(tf.Session().run(y)) # 将会输出 2
```

如果我想同时获得 y 和 z 的运算结果的话，只需将第 4 行改为如下代码即可：

```
04 print(tf.Session().run([y, z])) # 将会输出 [2, 3]
```

在本节的最后，我要指出一个非常容易犯错的地方：当使用了 `Variable` 时，必须调用初始化的方法之后，才能利用 `Session` 将相应的值从 Graph 里面提取出来。比如，下面这段代码是会报错的：

```
01 x = tf.Variable(1)
02 print(tf.Session().run(x)) # 报错！
```

应该改为：

```
01 x = tf.Variable(1)
02 with tf.Session().as_default() as sess:
03     sess.run(tf.global_variables_initializer())
04     print(sess.run(x))
```

其中 `tf.global_variables_initializer()` 的作用可由其名字直接得知：初始化所有 `Variable`。而如果只想初始化特定的 `Variable`，可以把第 3 行改成：

```
03 sess.run(tf.variables_initializer(var_list))
```

其中，`var_list` 就是我们想要初始化的 `Variable` 所组成的列表。

3.5.2 TensorFlow 模型的基本元素

3.5.1 节我们介绍了 TensorFlow 本身的组成单元，这一节我们会对 TensorFlow 模型的基本元素进行说明。虽说 TensorFlow 有很多种用法，不过一般而言我们都会拿它来实现基于梯度下降法的模型，所以 TensorFlow 模型的基本元素中大部分都是梯度下降模型的基本元素。换句话说，在我们的模型框架中至少应该包括：

- 数据生成器，它能够给我们提供批梯度下降法（MBGD）中的 Mini Batch。
- 样本权重，它衡量了各个样本相互之间的重要性关系，默认情况认为所有样本等权。
- 训练步数（`n_epoch`）以及一个训练步骤（`epoch`）中的迭代次数（`n_iter`）。
- 各种梯度下降算法所对应的优化器。
- 损失函数和评价模型好坏所用的指标。

需要指出的是，数据生成器的重要性是不言而喻的。虽然可能在小样本的情况下，数据生成器显得有些可有可无，但是当我们在面临实际问题时，数据的规模可能会非常大，在这种情况下，“分批”的思想就尤为重要了。我们在第3章讨论梯度下降算法时，曾经提过梯度下降算法有SGD（每次迭代只使用单一样本）、BGD（每次迭代都使用整个数据集）和MBGD算法（每次迭代中使用多个样本，通常会称这多个样本为一个Batch）。姑且不论MBGD和BGD算法本身孰优孰劣，单从内存问题来看，在数据量比较大时，BGD就不是一个可以接受的做法。因此通常采用的是MBGD算法，此时我们就需要将训练集“分批（Batch）”进行训练。

同样的道理，当数据量比较大时，直接在整个数据集上做预测也会引发内存不足的问题，为此同样需要分批来进行预测。不难看出，只要想在整个数据集上进行运算，我们就需要用到分批的思想，所以将分批的步骤单独抽象出来是很有必要的，而数据生成器恰好就是这样一个抽象。

那么接下来要做的，就是围绕上面这些基本元素来展开具体的代码实现。不过在此之前，为了保证这个框架的灵活性和可扩展性，我们需要整理一下实现的思路。

首先是模型超参数的管理。虽然我们能通过定义相应的TensorFlow Variable来自动化模型参数（比如权值矩阵、偏置量等）的训练，但模型超参数（比如神经网络的层数和每层神经元的个数）却还是需要我们来管理的。一种常见的做法是把“同一类型”的超参数放在一起进行管理，而机器学习模型的超参数可以大致分为如下两类：

- 与结构无关的超参数（后文统一简称为“模型超参数”）。
- 会影响到结构的参数（后文统一简称为“结构超参数”）。

我们会把这两类参数分别放在一个字典(dict)中进行管理，这样的话，只要把这两个字典保存下来，就能复现出整个模型，从而大大降低保存、复用的难度。

在知道了实现思路后，我们就可以来看看具体的初始化代码了，如代码3.1所示。

代码 3.1 基本框架的初始化：Base.py

```
01 class Base:
02     # 定义模型的“签名”
03     signature = "Base"
04
05     """
06     初始化结构
07     name: 模型的名字
08     model param settings: 管理“模型超参数”的字典
09     model structure settings: 管理“结构超参数”的字典
10     """
11     def __init__(self, name=None,
12                 model_param_settings=None, model_structure_settings=None):
13         # 定义一个字典，存储训练过程中的中间结果
14         self.log = {}
15         self._name = name
16         # 定义名字后缀，用于区分各种不同的模型
```

```

17     self.name appendix = ""
18     # 定义标识“是否已经初始化过超参数”的属性
19     self.settings initialized = False
20
21     # 定义数据生成器的种类
22     self.generator base = Generator
23     # 分别定义代指训练、交叉验证所用的数据生成器的属性
24     self.train generator = self.test generator = None
25     # 分别定义代指样本权重以及
26     # TensorFlow 计算图中的样本权重所对应的 placeholder 的属性
27     self.sample weights = self.tf sample weights = None
28     # 分别定义记录特征维度与类别数目的属性
29     # 当问题是回归问题时, self.n class 就会是 1
30     self.n dim = self.n class = None
31     # 定义进行 snapshot 时, 训练集与交叉验证集的抽样数
32     # snapshot 的定义会在下一节给出, 这里暂时按下不表
33     self.n random train subset = self.n random test subset = None
34
35     # 处理“模型超参数”
36     if model param settings is None:
37         self.model param settings = {}
38     else:
39         assert msg = "model param settings should be a dictionary"
40         assert isinstance(model param settings, dict), assert msg
41         self.model param settings = model param settings
42     self.lr = None
43     self.loss = self.loss name = self.metric name = None
44     self.optimizer name = self.optimizer = None
45     self.n epoch = self.max epoch = self.n iter = self.batch size = None
46
47     # 处理“结构超参数”
48     if model structure settings is None:
49         self.model structure settings = {}
50     else:
51         assert_msg = "model_structure_settings should be a dictionary"
52         assert isinstance(model_structure_settings, dict), assert_msg
53         self.model_structure_settings = model_structure_settings
54
55     # 定义一些辅助模型保存、复用的属性
56     # 在 3.5.4 节, 我们会给出模型保存与复用的具体实现和相关说明
57     self._model_built = False
58     self.py_collections = self.tf_collections = None
59     self._define_py_collections()
60     self._define_tf_collections()
61
62     # 定义模型参数相关的属性
63     # self._ws, self._bs 分别存储着模型的权值矩阵与偏置量
64     self._ws, self._bs = [], []
65     # self._is_training 将会标识当前是否是训练过程

```



```

66     self.is_training = tf.placeholder(tf.bool, name="is_training")
67     # self.loss 为模型算法的损失函数
68     # self.train_step 为相应的参数更新步骤
69     self.loss = self.train_step = None
70     # self.tfx, self.tfy 分别是模型输入中
71     # 特征向量和标签所对应的placeholder
72     # self.output, self.prob_output 则分别是模型的原始输出和概率输出
73     self.tfx = self.tfy = self.output = self.prob_output = None
74
75     # 定义一个代指 Session 的属性
76     self.sess = None
77     # 初始化一张 TensorFlow 的计算图 Graph
78     self.graph = tf.Graph()
79     # 从模型超参数中获取 Session 的设置
80     self.sess_config = self.model_param_settings.pop("sess_config", None)

```

以上我们就完成了基本框架的初始化，注意本节一开始就曾经说过，损失函数和评价模型好坏所用的指标是 TensorFlow 模型的基本元素，而这两者的计算都是可以抽象出来作为单独的类的。此外，神经网络的基本运算单元中的激活函数，其实也应该单独抽象出来以方便我们试用自己设计的激活函数。所以在具体搭建框架本身之前，我们需要先准备一些“辅助性的”内容。首先来看计算损失的类（Losses），如代码 3.2 所示。

代码 3.2 神经网络的辅助工具：NNUtil.py

```

01 # 导入相关的第三方库
02 import numpy as np
03 import tensorflow as tf
04 import scipy.stats as ss
05 from sklearn import metrics
06 # 定义能够根据模型预测 (pred) 与真值 (y) 来计算损失的类
07 class Losses:
08     # 定义 (欧氏) 距离损失函数
09     @staticmethod
10     def mse(y, pred, _, weights=None):
11         if weights is None:
12             return tf.losses.mean_squared_error(y, pred)
13         return tf.losses.mean_squared_error(y, pred, tf.reshape(weights, [-1, 1]))
14
15     # 定义交叉熵损失函数
16     # 注意我们需要根据 pred 是否已经是概率向量来调整损失的计算方法
17     @staticmethod
18     def cross_entropy(y, pred, already_prob, weights=None):
19         if already_prob:
20             # 如果 pred 已经是概率向量的话，就对其取对数，从而后面
21             # 计算 Softmax + Cross Entropy 时，就能还原出 pred 本身
22             eps = 1e-12
23             pred = tf.log(tf.clip_by_value(pred, eps, 1 - eps))
24         if weights is None:
25             return tf.losses.softmax_cross_entropy(y, pred)

```

```

26     return tf.losses.softmax_cross_entropy(y, pred, weights)
27
28     # 定义相关性系数损失函数
29     @staticmethod
30     def correlation(y, pred, weights=None):
31         # 利用 tf.nn.moments 算出均值与方差
32         y_mean, y_var = tf.nn.moments(y, 0)
33         pred_mean, pred_var = tf.nn.moments(pred, 0)
34         # 利用均值、方差与相应公式算出相关性系数
35         if weights is None:
36             e = tf.reduce_mean((y - y_mean) * (pred - pred_mean))
37         else:
38             e = tf.reduce_mean((y - y_mean) * (pred - pred_mean) * weights)
39         # 将损失设置为负相关性，从而期望模型输出与标签的相关性增加
40         return -e / tf.sqrt(y_var * pred_var)

```

需要指出的是，我们在计算 Cross Entropy 时，应该根据模型预测（pred）是否已经是概率向量来调整 Cross Entropy 的计算方法。这是因为，如果 pred 已经是概率向量而我们仍然强行用 Softmax 来把它概率化的话，从数学的角度来看就会显得非常矛盾（虽然最终结果可能不会差多少）。

接下来看看如何实现评估模型表现的类（Metrics）。需要指出的是，以下将会实现的、模型评价指标的较为详细的介绍会在附录 E 中给出，感兴趣的读者可以先行参阅相应部分。

```

41 # 定义能够根据模型预测（pred）与真值（y）来评估模型表现的类
42 class Metrics:
43     """
44     定义两个辅助字典
45     sign dict: key 为 metric 名, value 为±1, 其中
46         1: 说明该 metric 越大越好
47         -1: 说明该 metric 越小越好
48     require prob: key 为 metric 名, value 为 True 或 False, 其中
49         True: 说明该 metric 需要接受一个概率预测值
50         False: 说明该 metric 需要接受一个类别预测值
51     """
52     sign dict = {
53         "f1 score": 1,
54         "r2 score": 1,
55         "auc": 1, "multi auc": 1, "acc": 1, "binary acc": 1,
56         "mse": -1, "ber": -1, "log loss": -1,
57         "correlation": 1
58     }
59     require prob = {name: False for name in sign dict}
60     require prob["auc"] = True
61     require prob["multi auc"] = True
62
63     # 定义能够调整向量形状以适应相应 metric 函数的输入要求的方法
64     # 由于 scikit-learn 的 metrics 中定义的函数接收的参数都是一维数组
65     # 所以该方法的主要目的是把二维数组转为合乎要求的、相应的一维数组

```

```

66     @staticmethod
67     def check_shape(y, binary=False):
68         y = np.asarray(y, np.float32)
69         # 当 y 是二维数组时
70         if len(y.shape) == 2:
71             # 如果是二分类问题 (y=0 或 1)
72             if binary:
73                 # 而且 y 还是二维数组的话
74                 if y.shape[1] == 2:
75                     # 就返回第二列的相应预测值
76                     return y[:, 1]
77                 # 否则, 就把 y 铺平后返回
78                 return y.ravel()
79             # 如果不是二分类问题的话, 返回 y 的 argmax
80             return np.argmax(y, axis=1)
81         # 当 y 不是二维数组 (即 y 是一维数组) 时, 直接返回 y 即可
82         return y
83
84     # 定义计算 f1 score 的方法, 一般用于不平衡的二分类问题
85     @staticmethod
86     def f1_score(y, pred):
87         return metrics.f1_score(
88             Metrics.check_shape(y), Metrics.check_shape(pred)
89         )
90
91     # 定义计算 r2 score 的方法, 一般用于回归问题
92     @staticmethod
93     def r2_score(y, pred):
94         return metrics.r2_score(y, pred)
95
96     # 定义计算 auc 的方法, 也是一般用于不平衡的二分类问题
97     @staticmethod
98     def auc(y, pred):
99         return metrics.roc_auc_score(
100             Metrics.check_shape(y, True),
101             Metrics.check_shape(pred, True)
102         )
103
104     # 定义计算多分类 auc 的方法, 一般用于不平衡的多分类问题
105     @staticmethod
106     def multi_auc(y, pred):
107         n_classes = pred.shape[1]
108         if len(y.shape) == 1:
109             # 这里用到了 Toolbox 这个类, 其具体实现会放在本节最后
110             y = Toolbox.get_one_hot(y, n_classes)
111         fpr, tpr = [None] * n_classes, [None] * n_classes
112         for i in range(n_classes):
113             fpr[i], tpr[i], _ = metrics.roc_curve(y[:, i], pred[:, i])
114         new_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

```

```

115     new tpr = np.zeros like(new fpr)
116     for i in range(n_classes):
117         new tpr += interp(new fpr, fpr[i], tpr[i])
118     new tpr /= n_classes
119     return metrics.auc(new fpr, new tpr)
120
121     # 定义计算准确率的方法，一般用于均衡二分类问题与多分类问题
122     @staticmethod
123     def acc(y, pred):
124         return np.mean(Metrics.check_shape(y) == Metrics.check_shape(pred))
125
126     # 定义计算二分类准确率的方法
127     @staticmethod
128     def binary_acc(y, pred):
129         return np.mean(
130             (Metrics.check_shape(y) > 0) == (Metrics.check_shape(pred) > 0))
131
132     # 定义计算（欧氏）平均距离的方法，一般用于回归问题
133     @staticmethod
134     def mse(y, pred):
135         return np.mean(np.square(y.ravel() - pred.ravel()))
136
137     # 定义计算 log loss 的方法，一般用于分类问题
138     @staticmethod
139     def log_loss(y, pred):
140         return metrics.log_loss(y, pred)
141
142     # 定义计算相关性系数的方法，一般用于回归问题
143     @staticmethod
144     def correlation(y, pred):
145         return float(ss.pearsonr(y, pred)[0])

```

利用 `Metrics` 这个类的话，我们就能在基本框架中定义一个能根据指标的名字来返回计算相应指标的属性了：

```

01     @property
02     def metric(self):
03         return getattr(Metrics, self._metric_name)

```

接下来要做的，就是定义能够提供激活函数的类（`Activations`）。虽说在 TensorFlow 的 `tf.nn` 中已经帮我们实现了绝大多数的激活函数，但一些特殊的激活函数则可能仍然需要手动来实现：

```

146     # 定义能够根据输入 x 返回激活值的类
147     class Activations:
148         @staticmethod
149         def elu(x, name):
150             return tf.nn.elu(x, name)
151
152         @staticmethod

```

```

153     def relu(x, name):
154         return tf.nn.relu(x, name)
155
156     # 定义 selu 激活函数
157     # 对于 TensorFlow 1.4 以上的版本可以直接使用 tf.nn.selu
158     @staticmethod
159     def selu(x, name):
160         alpha = 1.6732632423543772848170429916717
161         scale = 1.0507009873554804934193349852946
162         return tf.multiply(scale, tf.where(x >= 0., x, alpha * tf.nn.elu(x)), name)
163
164     @staticmethod
165     def sigmoid(x, name):
166         return tf.nn.sigmoid(x, name)
167
168     @staticmethod
169     def tanh(x, name):
170         return tf.nn.tanh(x, name)
171
172     @staticmethod
173     def softplus(x, name):
174         return tf.nn.softplus(x, name)
175
176     @staticmethod
177     def softmax(x, name):
178         return tf.nn.softmax(x, name=name)
179
180     # 定义 sign 激活函数，它的应用场景会在下一章进行说明
181     @staticmethod
182     def sign(x, name):
183         return tf.sign(x, name)
184
185     # 定义 one_hot 激活函数，它的应用场景会在下一章进行说明
186     @staticmethod
187     def one_hot(x, name):
188         return tf.multiply(
189             x, tf.cast(tf.equal(
190                 x, tf.expand_dims(tf.reduce_max(x, 1), 1)
191             ), tf.float32),
192             name=name
193         )

```

注意：大家可能会想问：这三个类中的大多数方法无非是 `tf.losses`、`tf.nn` 和 `sklearn.me` 中相应方法的调用，为什么还要大费周折地额外弄三个类出来呢？这主要是为了提供一个统一的接口，从而使我们不仅能在实现神经网络模型时方便地调用它们，而且在想要扩展我们的模型（比如想要实现并实验一个新的损失函数）时提供便利——我们只需在相同的接口下进行实现即可，实验部分的代码则并不需要做出改变。

最后则是一些神经网络读取数据时可能用到的小工具，比如在第 1 章中提到过的从文件中将数据读入 Python 的方法，以及将 y 转化成 one-hot representation 的方法：

```

194 # 实现一些小工具
195 class Toolbox:
196     # 定义能够从分割符为 sep 的文件中读入数据的方法
197     @staticmethod
198     def get_data(file, sep=" ", include_header=False):
199         print("Fetching data")
200         data = [
201             [elem if elem else "nan" for elem in line.strip().split(sep)]
202             for line in file
203         ]
204         # 如果文件中有 header 的话，就要跳过第一行
205         if include_header:
206             return data[1:]
207         return data
208
209     # 定义能够将 y 转化成 one-hot representation 的方法
210     @staticmethod
211     def get_one_hot(y, n_classes):
212         if y is None:
213             return
214         one_hot = np.zeros([len(y), n_classes])
215         one_hot[range(len(one_hot)), np.asarray(y, np.int)] = 1
216         return one_hot

```

在本节的最后，我们来看一个稍微特殊一点的基本元素的实现，那就是“线性映射”的实现。之所以称线性映射为 TensorFlow 模型的基本元素，一是因为本书的主题是神经网络，而在神经网络的基本运算单元中，激活函数的实现已经在前文介绍过了，但线性映射的实现却还没介绍；另一个原因是 TensorFlow 本身是深度学习框架，所以它对矩阵乘法做了很充分的优化。即使我们拿它来实现其他模型，在大部分情况下都避不开线性映射。也正因此，TensorFlow 中已有相应的函数直接实现了线性映射。不过为了更好地设计网络结构，我们有必要将线性映射从头实现一遍（事实上我们在第 5 章介绍神经网络的剪枝时，就会将线性映射的过程做一些改动）：

```

01 # 定义线性映射过程，其中
02 # net 是线性映射的输入，shape 是其中权值矩阵的 shape
03 # appendix 则是线性映射这一套运算步骤所属的 name_scope 的后缀
04 def _fully_connected_linear(self, net, shape, appendix):
05     with tf.name_scope("Linear{}".format(appendix)):
06         # 调用 init_w、init_b 方法来初始化权值矩阵与偏置量
07         w = init_w(shape, "W{}".format(appendix))
08         b = init_b([shape[1]], "b{}".format(appendix))
09         # 用 self._ws 和 self._bs 记录下这两个参数
10         self._ws.append(w)
11         self._bs.append(b)
12         # 返回线性映射的结果

```

```

13         return tf.add(
14             tf.matmul(net, w),
15             b, name="Linear{}_Output".format(appendix))

```

其中 `init_w` 和 `init_b` 这两个方法的实现是定义在 `NNUtil.py` 中的，在看它们的具体代码之前，我们先要知道从理论上应该如何进行参数的初始化。

首先来看权值矩阵，它的初始化方式有很多种，比如均匀分布采样（https://www.tensorflow.org/api_docs/python/tf/random_uniform）、正态分布采样（https://www.tensorflow.org/api_docs/python/tf/random_normal）和截断正态分布采样（https://www.tensorflow.org/api_docs/python/tf/truncated_normal）等，我们采用的初始化方法则是一种比较特殊的截断正态分布采样——Xavier Initialization（<http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization>）。

注意：Xavier Initialization 不要求一定用截断正态分布采样，它也可以用正态分布采样。

是否进行截断的区别可以参见 <https://stackoverflow.com/questions/41704484/what-is-difference-between-tf-truncated-normal-and-tf-random-normal>，这里从略。

所谓的 Xavier Initialization，就是均值为 0、方差为 $\text{Var}(\mathbf{W})$ 的（截断）正态分布采样。其中，参数 \mathbf{W} 的方差需要满足：

$$\text{Var}(\mathbf{W}) = \frac{1}{n_{\text{in}}}$$

或

$$\text{Var}(\mathbf{W}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

公式里的 n_{in} 和 n_{out} 分别代指权值矩阵 \mathbf{W} 所对应的输入、输出神经元个数。本书将采用第二个公式来进行实现，相应的代码非常直观：

```

01 # 参数 shape 的第 0 位、第 1 位即为  $n_{\text{in}}$ 、 $n_{\text{out}}$ 
02 def init_w(self, shape, name):
03     return tf.Variable(tf.truncated_normal(
04         shape, stddev=math.sqrt(2/sum(shape)))
05     ), name=name)

```

唯一需要注意的是，由于 `tf.truncated_normal` 接受的参数 `stddev` 的意义是标准差，所以我们要额外开一个根号。

然后来看看偏置量的初始化，这个倒是没有特殊的技巧，我们既可以均匀分布采样也可以直接初始化为零向量。考虑到第 2 章介绍感知机时提到过的 Novikoff 定理在偏置量初始化为零向量时相对而言比较简单，所以我们这里就直接依样画葫芦：

```

01 # 参数 shape 即为当前层的神经元个数
02 def init_b(self, shape, name):
03     return tf.Variable(tf.zeros(shape), name=name)

```

此外还需要注意的是，我们在定义线性映射时，返回的结果是

```
return tf.add(tf.matmul(net, w), b, name="Linear{}_Output".format(appendix))
```

这意味着在实际操作中，线性映射的公式其实是

$$\mathbf{h}^{(i)} = \mathbf{W}_0^{(i)} + \mathbf{O}^{(i-1)}\mathbf{W}^{(i)}$$

这是因为我们在使用梯度下降法时一般使用的都是 MBGD，从而 \mathbf{O} 一般会是一个 M 个特征向量 \mathbf{o} 的堆叠。由于从直观上来说，堆叠的方式一般是从下往上堆，此时特征向量 \mathbf{o} 将被视为行向量，从而 \mathbf{O} 一般就会是 $M \times n^{(i-1)}$ 的矩阵，其中 M 是 Mini Batch 的大小。当然，如果想要和推导中的公式保持一致的话，只需让 \mathbf{O} 是 M 个 \mathbf{o} 从左往右堆叠所得到的矩阵即可。不过对于本书而言，我们还是倾向于使用更直观的实现方式，此时 $\mathbf{W}^{(i)}$ 将会是一个 $n^{(i-1)} \times n^{(i)}$ 的矩阵，即它是公式推导中对应的权值矩阵的转置。

以上就是实现神经网络所需的一些辅助性工具，接下来我们会利用它们以及之前初始化好的超参数，来将 TensorFlow 模型基本框架中的各个部分给填补齐全。

3.5.3 TensorFlow 元素的整合方法

在这一节中，我们会先介绍基本框架的核心方法——fit 方法的实现，该方法会整合所有需要用到的 TensorFlow 元素并完成模型的训练。由于 fit 方法是核心中的核心，所以相应的代码也会比较长，为此我们分两部分来介绍。第一部分是准备部分，fit 方法会利用初始化好的超参数来进行各种设置；第二部分则是核心循环部分，TensorFlow 模型的训练将会在这个循环中全部完成。

首先来看看 fit 方法中的准备部分，如代码 3.3 所示。

代码 3.3 基本框架的训练：Base.py

```
01  """
02      基本框架的训练
03      x, y: 训练所用的特征向量和标签
04      x_test, y_test: 交叉验证所用的特征向量和标签
05          取 None 则意味着不进行交叉验证
06      sample_weights: 样本权重，当取 None 时意味着认为所有样本等权
07      names: 训练集与交叉验证集的名字
08          在短时间之内我们都不会用到它，不过在第 6 章时将看到其重要性
09      timeit: 是否计时
10      snapshot_ratio: 一个 epoch 中进行快照 (snapshot) 的次数
11          如果设为 0 的话，就认为：
12              1) 一个 epoch 中只进行一次 snapshot
13              2) 不使用训练监控器 (monitor)
14      print_settings: 是否 print 出模型的各种设置
15      verbose: 控制着是否输出中间结果，以及是否进行 Tensorboard 可视化
16          verbose = 0: 既不输出中间结果，也不进行 Tensorboard 可视化
17          verbose = 1: 输出中间结果，但不进行 Tensorboard 可视化
18          verbose = 2: 输出中间结果，且进行 Tensorboard 可视化
```



```

19     """
20     def fit(self, x, y, x_test=None, y_test=None, sample_weights=None,
21             names=("train", "test"), timeit=True, snapshot_ratio=3,
22             print_settings=True, verbose=1):
23         # 如果要计时的话, 就记录下当前的时间
24         t = None
25         if timeit:
26             t = time.time()
27
28         # 利用输入的数据进行模型各种超参数的初始化, 这里面涉及的
29         # self.init from data 和 self.init all settings 的具体说明会马上在后文给出
30         self.init from data(x, y, x_test, y_test, sample_weights, names)
31         if not self.settings_initialized:
32             self.init all settings()
33         self.settings_initialized = True
34
35         # 如果还没有搭建模型
36         # 就调用相应的方法在计算图 self.graph 中搭建网络结构
37         if not self.model_built:
38             with self.graph.as_default():
39                 # 定义模型的输入与 placeholder
40                 with tf.name_scope("Input"):
41                     self.define_input_and_placeholder()
42                 # 搭建模型的结构
43                 with tf.name_scope("Model"):
44                     self.build_model()
45                     self.prob_output = tf.nn.softmax(
46                         self.output, name="Prob Output")
47                 # 根据初始化超参数时获得的损失函数和优化器来具体地定义出
48                 # 模型的损失以及相应的参数更新步骤
49                 with tf.name_scope("LossAndTrainStep"):
50                     self.define_loss_and_train_step()
51                 # 初始化 TensorFlow 的 Variable
52                 with tf.name_scope("InitializeVariables"):
53                     self._initialize_variables()
54
55                 # 初始化 epoch、iter 和 snapshot 的指针
56                 i_epoch = i_iter = j = snapshot_cursor = 0
57                 # 进行快照 (snapshot) 的相关设置
58                 # 注意只有在提供了交叉验证集时, 才能使用 monitor
59                 if snapshot_ratio == 0 or x_test is None or y_test is None:
60                     use_monitor = False
61                     snapshot_step = self.n_iter
62                 else:
63                     use_monitor = True
64                     snapshot_step = int(self.n_iter / snapshot_ratio)
65
66                 # 初始化训练的各种监控指标
67                 terminate = False

```

```

68         over fitting flag = 0
69         n epoch = self.n epoch
70         # 初始化模型参数的临时缓存路径
71         tmp checkpoint folder = os.path.join(self.model saving path, "tmp")
72         # 初始化训练监控器 monitor, 其具体实现的说明会放在第 6 章
73         # 该监控器能够监控训练, 它能做到如下两点:
74         # 1) 若发现过拟合则提前停止训练
75         # 2) 若发现欠拟合则延长训练步数
76         # TrainMonitor 是非常重要的工具, 它能让我们免于调整训练步长
77         # 从而能让我们把注意力集中在模型本身的调优上面
78         monitor = TrainMonitor(
79             Metrics.sign dict[self. metric name], snapshot ratio
80         ).start new run()
81
82         if verbose >= 2:
83             prepare tensorboard verbose(self. sess)
84
85         if print settings:
86             self.print settings()
87
88         # 定义存储中间结果的列表
89         self.log["iter loss"] = []
90         self.log["epoch loss"] = []
91         self.log["test snapshot loss"] = []
92         self.log["train {}".format(self. metric name)] = []
93         self.log["test {}".format(self. metric name)] = []
94         # 对模型的初始结果做一个 snapshot
95         # 相关方法的定义会放在本节最后
96         self._snapshot(0, 0, 0)

```

这段代码中涉及许多需要补充的东西。首先是 `self.init_from_data` 方法, 该方法其实对应着前文所说的数据预处理函数 $\phi^{(0)}$ 。作为一个基本框架, 它需要做的工作并不多, 大部分都只是利用数据生成器来获得各种属性 (数据生成器的具体实现会在附录 F 中进行说明, 这里暂不赘述。此外, 虽然我们传了一个 `names` 的参数到该方法中, 但正如前文所说, 这个参数暂时还不会被用到。直到第 6 章时, 该参数的作用才会被显现出来):

```

01     def init_from_data(self, x, y, x_test, y_test, sample_weights, names):
02         # 处理样本权重
03         self._sample_weights = sample_weights
04         if self._sample_weights is None:
05             # 如果没有提供样本权重的话
06             # 就没有必要定义相应的占位符 (placeholder)
07             self._tf_sample_weights = None
08         else:
09             # 否则, 就需要定义相应的 placeholder 以供后面搭建模型时使用
10             self._tf_sample_weights = tf.placeholder(
11                 tf.float32, name="sample_weights")
12
13         # 根据 x、y 和样本权重来获取训练数据的数据生成器

```

```

14     self.train_generator = self.generator_base(
15         x, y, "TrainGenerator", self.sample_weights, self.n_class)
16     # 如果提供了交叉验证集, 则获取相应的数据生成器
17     if x_test is not None and y_test is not None:
18         self.test_generator = self.generator_base(
19             x_test, y_test, "TestGenerator", n_class=self.n_class)
20     else:
21         self.test_generator = None
22     # 计算获取中间结果时所用的样本数量
23     # 对于训练数据而言, 由于样本总量可能很大, 所以只取 10%
24     # 对于交叉验证的数据而言, 为使结果更具统计意义, 所以就取全部样本
25     self.n_random_train_subset = int(len(self.train_generator) * 0.1)
26     if self.test_generator is None:
27         self.n_random_test_subset = -1
28     else:
29         self.n_random_test_subset = int(len(self.test_generator))
30
31     # 根据数据生成器来获取特征维度与类别数目
32     # 这里的特征维度严格意义上来说, 其实是“连续型特征的个数”
33     # 不过由于我们实现的是朴素神经网络, 所以暂时可以直接认为
34     # 所有特征都是连续型特征
35     self.n_dim = self.train_generator.shape[-1]
36     self.n_class = self.train_generator.n_class
37
38     # 初始化每个 Mini Batch 中的样本量, 默认为 128
39     self.batch_size = self.model_param_settings.get("batch_size", 128)
40     # 如果总样本量都没有 self.batch_size 多, 就用 BGD 代替 MBGD
41     self.batch_size = min(self.batch_size, len(self.train_generator))
42     # 初始化每一个 epoch 中的迭代次数
43     self.n_iter = self.model_param_settings.get("n_iter", -1)
44     if self.n_iter < 0:
45         # 如果没有指定的话, 就将迭代次数设置为
46         # 一个 epoch 中能把所有训练样本都过一遍
47         self.n_iter = int(len(self._train_generator) / self.batch_size)

```

其次是 `self.init_all_settings` 方法, 该方法能够初始化所有模型超参数, 但它本身其实只是两个子方法的结合:

```

01     def init_all_settings(self):
02         # 初始化所有“模型超参数”
03         self.init_model_param_settings()
04         # 初始化所有“结构超参数”
05         self.init_model_structure_settings()

```

所以关键还在于 `self.init_model_param_settings` 和 `self.init_model_structure_settings` 这两个初始化方法的实现。在给出具体的代码之前, 我们先来看看实现的逻辑。

首先是何要把模型超参数和结构超参数分别放在一个字典中来管理。除了前文说过的容易复现的理由之外, 还有一个重要的理由其实是, 这样能够极大地增强模型的灵活性。如果我

们不把这些超参数放进字典而把它们直接放在函数参数中的话，当我们不断地加入新功能时，函数的参数就会变得越来越长，这样不仅非常不优雅，而且会给维护带来极大的困难。反之，如果把这些参数放进字典中，那么我们就只需维护这个字典，对函数的参数则基本不需要过多进行维护。这样会使代码看上去更优雅的同时，大大降低维护的难度。

其次就是怎样具体地进行管理的问题，为此我们需要频繁地用到 Python 中字典这个数据结构的 `get` 方法，下面就来看一个小例子。

```
# 定义一个字典，a 对应 1，b 对应 2
d = {'a': 1, 'b': 2}
# get 方法的用处是：如果字典中有目标的 key，那么返回相应的 value
# 否则，返回 None 或是给定的一个值
print(d.get('a')) # 将会输出 1
print(d.get('c')) # 将会输出 None
print(d.get('c', 3)) # 将会输出 3
```

除了 `get` 方法以外，我们还会频繁用到 `getattr` 这个函数。`getattr` 是 `get attribute` 的缩写，作用事实上也是 `get attribute`（获取属性）：它会接收一个类或实例和一个字符串来分别作为第一个和第二个参数，然后将它们传入类或实例中，名字为传入的字符串的属性输出。比如，现在有一个实例，名为 `student`，它有一个属性叫 `name`，那么我们就能通过

```
getattr(student, "name")
```

来获取到这个属性（注意 `getattr` 接收的第二个参数是字符串）。

在有了 `get` 和 `getattr` 方法后，我们就能很轻松地进行相应超参数的初始化了：

```
06 # 初始化模型超参数
07 def init_model_param_settings(self):
08     # 定义模型的损失函数
09     loss = self.model_param_settings.get("loss", None)
10     if loss is None:
11         # 如果没有指定的话就使用默认的配置，具体而言：
12         # 分类问题使用 Softmax + Cross Entropy，回归问题使用相关性系数
13         if self.n_class == 1:
14             self.loss_name = "correlation"
15         else:
16             self.loss_name = "cross entropy"
17     else:
18         self.loss_name = loss
19     # 定义模型的评价指标
20     metric = self.model_param_settings.get("metric", None)
21     if metric is None:
22         # 如果没有指定的话就使用默认的配置，具体而言：
23         # 回归问题使用相关性系数，二分类、多分类问题则分别使用
24         # 二分类对应的 auc 指标与多分类对应的 auc 指标
25         if self.n_class == 1:
26             self.metric_name = "correlation"
27         elif self.n_class == 2:
```

```

28         self.metric name = "auc"
29     else:
30         self.metric name = "multi auc"
31     else:
32         self.metric name = metric
33     # 初始化训练步数, 默认为 32
34     self.n epoch = self.model param settings.get("n epoch", 32)
35     # 初始化最大训练步数, 默认为 256
36     self.max epoch = self.model param settings.get("max epoch", 256)
37     # 若 self.n epoch 比 self.max epoch 大
38     # 就将 self.max epoch 扩大至 self.n epoch
39     self.max epoch = max(self.max epoch, self.n epoch)
40
41     # 由于之前已经把 batch size 和 n iter 初始化了
42     # 所以这里直接读取即可
43     self.batch size = self.model param settings["batch size"]
44     self.n iter = self.model param settings["n iter"]
45
46     # 初始化优化器和训练速率, 默认使用 Adam 和 0.001
47     self.optimizer name = self.model param settings.get(
48         "optimizer", "Adam")
49     self.lr = self.model param settings.get("lr", 1e-3)
50     self.optimizer = getattr(tf.train, "{}Optimizer".format(
51         self.optimizer name))(self.lr)
52
53     # 初始化结构超参数。由于我们这里实现的是基本框架
54     # 所以无须定义结构, 从而也无须初始化结构超参数
55     def init model structure settings(self):
56         pass

```

以上我们就完成了各种超参数的初始化, 下面来看看代码 3.3 的第 40 到 53 行是如何利用这些初始化好的超参数来进行各种模型结构的设置的。对于框架本身而言, 这些设置的实现要么比较容易, 要么暂时不需要进行具体实现:

```

01     def define input and placeholder(self):
02         # 将标识是否处在训练过程的属性设置为对应的 placeholder
03         self.is training = tf.placeholder(tf.bool, name="is training")
04         # 根据特征维度 self.n dim 和类别数 self.n class
05         # 来定义模型输入中特征向量和标签所对应的 placeholder
06         self.tfx = tf.placeholder(tf.float32,
07             shape=[None, self.n dim], name="X")
08         self.tfy = tf.placeholder(tf.float32,
09             shape=[None, self.n class], name="Y")
10
11         # 模型搭建的方法留待后续补充, 对于框架本身则无须实现它
12         def build model(self, net=None):
13             pass
14
15         # 利用损失函数名和优化器来定义出损失函数和参数更新步骤

```

```

16     def define_loss_and_train_step(self):
17         self.loss = getattr(Losses, self.loss_name)(
18             self.tfy, self.output, False, self.tf_sample_weights
19         )
20         # 利用 TensorFlow 的方法来定义出参数更新步骤 self.train_step
21         with tf.control_dependencies(tf.get_collection(
22             tf.GraphKeys.UPDATE_OPS
23         )):
24             self.train_step = self.optimizer.minimize(self.loss)
25
26         # 利用计算图 (self.graph) 和相应的设置 (self.sess_config)
27         # 来初始化一个 TensorFlow 的 Session
28         def initialize_session(self):
29             self.sess = tf.Session(graph=self.graph, config=self.sess_config)
30
31         # 定义 TensorFlow 变量 (Variables) 的初始化步骤
32         def initialize_variables(self):
33             self._sess.run(tf.global_variables_initializer())

```

在上述代码的第 1 行到第 9 行处, 我们第一次使用了 `placeholder` 这个 TensorFlow 中极其重要的基本元素。顾名思义, 它是 TensorFlow 运算图 Graph 中的一个“占位符”, 能够描述“不固定”的数据。事实上在代码中, 我们把一个 `None` 传入了 `placeholder` 的 `shape` 参数, 这说明对应的 `placeholder` 的长度是未定的, 而这就是 TensorFlow 灵活性的重要来源之一。具体到 MBGD 算法而言, 这意味着 Mini Batch 的大小可以是未定的。

下面来看一个简单的例子:

```

01 # 定义一个数据类型为 tf.float32、“长”未知、“宽”为 2 的矩阵的 placeholder
02 x = tf.placeholder(tf.float32, [None, 2])
03 # 定义一个 numpy 数组: [[ 1  2 ], [ 3  4 ], [ 5  6 ]]
04 y = np.array([[1, 2], [3, 4], [5, 6]])
05 # 定义 x + 1 对应的 Tensor
06 z = x + 1
07 # 利用 Session 及其 feed_dict 参数、将 y 的值赋给 x、同时输出 z 的值
08 print(tf.Session().run(z, feed_dict={x: y})) # 将会输出 [[ 2  3 ], [ 4  5 ], [ 6  7 ]]

```

于是, 我们分 Batch 训练或计算中间结果的实现步骤就很清晰了:

- 把损失函数对应的参数更新步骤或计算中间结果所涉及的所有变量都定义为占位符 `placeholder`。
- 每次训练时, 通过 `feed_dict` 参数将原数据的一个小 Batch 对应的值赋给这些 `placeholder`, 从而完成这个小 Batch 上的训练或计算出这个小 Batch 上的中间结果。

占位符还有许多其他有趣的应用方法, 不过它们的思想都是相通的: 将未能确定的信息以 `placeholder` 的形式进行定义, 然后在确实调用到的时候再赋予具体的数值。此外, 在具体的应用中, 由于可能会用到许许多多的 `placeholder`, 所以一般而言把获取 `feed_dict` 的过程抽象成一个单独的方法是一个良好的习惯, 在我们的实现中也会这样做。

此外,注意到代码 3.3 的第 45 行和第 46 行处,我们在执行了 `self._build_model` 这个方法后,马上就利用了 `Softmax` 函数并通过

```
self._prob_output = tf.nn.softmax(self._output, name="Prob_Output")
```

来获取模型的概率输出,这意味着我们后续利用该基本框架来搭建模型时,需要在 `self._build_model` 中定义好模型的原始输出 `self._output`。此外需要指出的是,在定义模型损失时,我们是用模型的原始输出 `self._output` 来定义的,这是因为 TensorFlow 许多自带的损失函数接收的输入都需要模型的原始输出而非概率输出。之所以额外定义了模型的概率输出 `self._prob_output`, 主要是为了方便快速地获取模型在交叉验证集上的概率输出,从而能够更快地进行模型的评估(因为有许多重要指标,比如 `auc`, 是要求使用概率输出来计算的)。

接下来看看 `fit` 方法的核心循环部分:

```
57     # 当训练步数小于总训练步数时就不断循环
58     while i_epoch < n_epoch:
59         i_epoch += 1
60         epoch_loss = 0
61         # 在一个 epoch 中迭代 self.n_iter 次
62         for j in range(self.n_iter):
63             i_iter += 1
64             # 利用 self._gen_batch 方法来获取一个 Mini Batch
65             x_batch, y_batch, sw_batch = self._gen_batch(
66                 self._train_generator, self.batch_size, one_hot=True
67             )
68             # 调用 self._sess.run 来完成训练并获取当前迭代的损失 iter_loss
69             # 同时,调用了 self._get_feed_dict 方法来获取当前的 feed_dict
70             iter_loss = self._sess.run(
71                 [self._loss, self._train_step],
72                 self._get_feed_dict(x_batch, y_batch, sw_batch,
73                                     is_training=True)
74             )[0]
75             self.log["iter_loss"].append(iter_loss)
76             epoch_loss += iter_loss
77             # 如果要进行 snapshot 的话,就调用相应方法进行 snapshot
78             if i_iter % snapshot_step == 0 and verbose >= 1:
79                 snapshot_cursor += 1
80                 train_metric, test_metric = self._snapshot(
81                     i_epoch, i_iter, snapshot_cursor
82                 )
83                 # 如果使用 monitor 且提供了交叉验证集的话
84                 # 就用它来对训练进行监控
85                 if use_monitor:
86                     check_rs = monitor.check(test_metric)
87                     over_fitting_flag = monitor.over_fitting_flag
88                     # 如果 monitor 给出了“终止”的信息,就提前终止训练
89                     if check_rs["terminate"]:
90                         n_epoch = i_epoch
91                         terminate = True
```

```

92             Break
93             # 如果 monitor 给出了“保存 checkpoint”的信息
94             # 就将当前模型参数保存下来以便回滚
95             if check rs["save checkpoint"]:
96                 print(" - {}".format(check rs["info"]))
97                 self.save_checkpoint(tmp_checkpoint_folder)
98 self.log["epoch loss"].append(epoch_loss / (j + 1))
99 # 如果使用了 monitor, 就要额外地进行一些操作
100 if use_monitor:
101     # 如果达到了训练步数, 但是 monitor 没有给出“终止”的信息,
102     # 而且训练步数没达到预先设置的最大步数 max_epoch 的话,
103     # 就延长训练步数, 同时对延长训练步数做一个惩罚
104     if i_epoch == n_epoch:
105         if i_epoch < self.max_epoch and not monitor.info["terminate"]:
106             monitor.flat_flag = True
107             monitor.punish_extension()
108             n_epoch = min(
109                 n_epoch + monitor.extension,
110                 self.max_epoch
111             )
112             print(" - Extending n_epoch to {}".format(n_epoch))
113     # 如果达到了最大训练步数
114     # 但是 monitor 仍然没给出“终止”的信息的话
115     if i_epoch == self.max_epoch:
116         terminate = True
117         if not monitor.info["terminate"]:
118             # 如果 monitor 认为还没有过拟合, 那么就提示用户
119             # 可以延长训练步数
120             if not over_fitting_flag:
121                 print(
122                     " - Model seems to be under-fitting"
123                     " but max epoch reached. Increasing"
124                     " max epoch may improve performance"
125                 )
126             # 否则, 就单纯地提示达到了最大步数
127             else:
128                 print(" - max_epoch reached")
129 elif i_epoch == n_epoch:
130     terminate = True
131 if terminate:
132     # 如果训练需要终止, 就将参数回滚到
133     # 当前已有的(最好的)checkpoint 上
134     if os.path.isdir(tmp_checkpoint_folder):
135         print(" - Rolling back to the best checkpoint")
136         self.restore_checkpoint(tmp_checkpoint_folder)
137         shutil.rmtree(tmp_checkpoint_folder)
138     break
139 # 把模型的最终表现 print 出来
140 self._snapshot(-1, -1, -1)

```



```

141
142     # 如果需要计时, 就把训练总时长 print 出来
143     if timeit:
144         print(" - Time Cost: {}".format(time.time() - t))
145
146     return self

```

在上面这段代码中, 第 72 行调用的 `self.get_feed_dict` 就是前文曾经提到过的、用来获取 `feed_dict` 的方法。对于目前而言, 由于我们只定义了模型的输入, 所以该方法目前也只需把模型的输入所涉及的 `placeholder` 所需的 `feed_dict` 给构造出来:

```

01     def get_feed_dict(self, x, y=None, weights=None, is_training=False):
02         # 赋予特征向量和是否正在训练的标识所对应的 placeholder 以具体的值
03         feed_dict = {self.tfx: x, self.is_training: is_training}
04         # 如果提供了 y 的话, 就赋予标签所对应的 placeholder 以具体的值
05         if y is not None:
06             feed_dict[self.tfy] = y
07         # 如果有样本权重的话, 就赋予样本权重所对应的 placeholder 以具体的值
08         if self.tf_sample_weights is not None:
09             if weights is None:
10                 weights = np.ones(len(x))
11             feed_dict[self.tf_sample_weights] = weights
12         return feed_dict

```

除了 `self.get_feed_dict` 这个重要方法外, 我们还在代码的第 65 行和第 80 行中分别调用了 `self.gen_batch` 和 `self.snapshot` 这两个方法。在这里, 我们大致介绍一下它们的具体实现。首先来看看 `self.gen_batch` 对应的代码:

```

13     # 利用数据生成器 generator 来生成一个 batch
14     def gen_batch(self, generator, n_batch,
15                  gen_random_subset=False, one_hot=False):
16         # 如果 gen_random_subset 参数为真, 则调用 gen_random_subset 方法
17         if gen_random_subset:
18             data, weights = generator.gen_random_subset(n_batch)
19         # 否则, 调用 gen_batch 方法
20         else:
21             data, weights = generator.gen_batch(n_batch)
22         x, y = data[..., :-1], data[..., -1]
23         # 如果不要求返回 OneHot, 直接返回结果即可
24         if not one_hot:
25             return x, y, weights
26         # 否则, 根据类别数来对 y 进行处理后, 再返回结果
27         if self.n_class == 1:
28             y = y.reshape([-1, 1])
29         else:
30             y = Toolbox.get_one_hot(y, self.n_class)
31         return x, y, weights

```

可以看到, 它基本只是数据生成器的各种应用, 所以如果想要理解里面的细节的话, 可能

还需要结合附录 F 中介绍数据生成器实现的相应部分。

至于 `self_snapshot` 方法的实现，它是依赖于上述 `self_gen_batch` 方法的，同时它还会用到模型的预测方法。毕竟 `snapshot` 的目的是做一个快照，所以我们需要把当前模型的表现给计算出来，这就要求我们对样本进行预测。

为了能让大家更关注于模型本身而不分心于实现细节，笔者实现了一个叫 `self_calculate` 的方法，它能利用数据计算出某个 `Tensor` 或某些 `Tensor` 的具体数值，而且在计算的过程中不用担心引发内存问题。`self_calculate` 方法的实现我们同样会在附录 F 中介绍，这里我们就直接应用它来实现模型的预测方法：

```
01     def predict(self, x):
02         # 利用数据 x 来计算出模型的输出
03         # 如果是回归问题，就获取原始输出 self.output
04         # 如果是分类问题，就获取概率输出 self.prob_output
05         tensor = self.output if self.n_class == 1 else self.prob_output
06         # 由于是预测过程，所以 is_training 参数要置为 False
07         output = self.calculate(x, tensor=self.prob_output, is_training=False)
08         # 如果是回归问题的话，就将输出展平
09         if self.n_class == 1:
10             return output.ravel()
11         # 否则，直接输出结果即可
12         return output
```

在有了预测的方法后，结合生成 `batch` 的方法，`self_snapshot` 的实现就比较自然了：

```
01     def snapshot(self, i_epoch, i_iter, snapshot_cursor):
02         # 获取 snapshot 所用的训练集
03         x_train, y_train, sw_train = self.gen_batch(
04             self.train_generator, self.n_random_train_subset,
05             gen_random_subset=True
06         )
07         # 如果提供了交叉验证集的话，就获取 snapshot 所用的交叉验证集
08         if self.test_generator is not None:
09             x_test, y_test, sw_test = self.gen_batch(
10                 self.test_generator, self.n_random_test_subset,
11                 gen_random_subset=True
12             )
13         else:
14             x_test = y_test = sw_test = None
15         # 获取模型在所选训练集与测试集（如果有的话）上的预测
16         y_train_pred = self.predict(x_train)
17         if x_test is not None:
18             # 如果是回归问题，就获取原始预测
19             if self.n_class == 1:
20                 tensor = self.output
21             # 否则，就获取概率预测
22             else:
23                 tensor = self._prob_output
```

```

24         y_test_pred, test_snapshot_loss = self.calculate(
25             x_test, y_test, sw_test,
26             [tensor, self.loss], is_training=False
27         )
28         # 如果是回归问题, 就将 y_test 给 reshape 成列向量
29         if self.n_class == 1:
30             y_test = y_test.reshape([-1, 1])
31         # 如果是分类问题, 就将 y_test 转化成 OneHot 向量
32         else:
33             y_test = Toolbox.get_one_hot(y_test, self.n_class)
34         y_test_pred = y_test_pred[0]
35         test_snapshot_loss = test_snapshot_loss[0]
36     else:
37         y_test_pred = test_snapshot_loss = None
38         # 利用 self.metric 这个 property 算出模型在训练集与测试集上的指标
39         train_metric = self.metric(y_train, y_train_pred)
40         if y_test is not None and y_test_pred is not None:
41             test_metric = self.metric(y_test, y_test_pred)
42             if i_epoch >= 0 and i_iter >= 0 and snapshot_cursor >= 0:
43                 self.log["test_snapshot_loss"].append(
44                     test_snapshot_loss)
45                 self.log["test {}".format(self.metric_name)].append(
46                     test_metric)
47                 self.log["train {}".format(self.metric_name)].append(
48                     train_metric)
49         else:
50             test_metric = None
51         # 输出相应的信息
52         print("\rEpoch {:6}   Iter {:8}   Snapshot {:6} ({} ) - "
53             "Train : {:8.6}   Test : {}".format(
54                 i_epoch, i_iter, snapshot_cursor,
55                 self.metric_name, train_metric,
56                 "None" if test_metric is None else "{:8.6}".format(test_metric)
57             ), end="")
58         # 如果是第一个 snapshot, 就额外多输出一个空行以保留初始指标
59         if i_epoch == i_iter == snapshot_cursor == 0:
60             print()
61         # 返回相应指标
62         return train_metric, test_metric

```

以上我们就完成了 TensorFlow 各种基本元素的整合, 并实现模型的训练步骤。接下来还需要做的, 就只剩下实现模型的保存 (save) 与复用 (load) 了。

3.5.4 TensorFlow 模型的 save & load

由于我们在初始化模型参数时通常会采用随机初始化, 所以即使是在同一组超参数下训练同样的训练步数, 得到的最终模型可能也会迥然不同。为此, 实现模型的保存与复用是非常有必要的一件事, 它不仅能让我们记录模型的中间训练结果以便过拟合后进行回滚, 也能让我们

将模型部署到不同的机器上。

TensorFlow 中各个 Tensor 的保存是比较简单的，这是因为它内部有一个相应的实现，能够把某个 Session 中的已经被初始化的 Variable 以及它们和其余 Tensor 的所有运算逻辑都存储在文件中。事实上，我们只需一行代码就能实现该功能：

```
tf.train.Saver().save(sess, folder)
```

其中，sess 是我们想要保存的 Session，folder 是我们想要将模型保存到的文件夹。不过放到实际中来看的话，不仅需要把 Tensor 本身保存下来，还需要把指向 Tensor 和 placeholder 的指针保存下来。否则，即使基本框架恢复出了所有的 Tensor 和运算逻辑，也无法具体调用它们。更麻烦的是，由于 placeholder 需要我们显式地给它赋值，否则相应的运算就无法进行；同时注意到之前定义的模型的入口 self.tfx 就是一个 placeholder，所以如果我们的框架连对应 placeholder 的属性都没有的话，那么整个模型就会连第一步的运算都无法进行。

万幸的是，TensorFlow 中同样有相应的、非常简易的实现来让我们保存指向 Tensor 和 placeholder 的指针。放在我们的基本框架中的话，代码如下所示：

```
01 def add_tf_collections(self):
02     # 把我们所有想要保存的指针对应的属性的名字都
03     # 放在 self.tf_collections 中，然后通过前文介绍过的 getattr 方法
04     # 以及 TensorFlow 中的 add_to_collection 方法，来完成这些
05     # 指向 Tensor 和 placeholder 的指针的保存
06     for tensor in self.tf_collections:
07         target = getattr(self, tensor)
08         if target is not None:
09             # 将 target 这个指针以 tensor 的名字保存下来
10             tf.add_to_collection(tensor, target)
```

其中，代码第 6 行处用到的 self.tf_collections 储存着各个指针。对于一个基本框架而言，我们关心的 Tensor 与 placeholder 有如下四类：

- 模型输入中，特征向量和标签对应的 placeholder (self.tfx、self.tfy)。
- 模型输出对应的 Tensor (self.output)。
- 模型训练对应的 Tensor (self.loss、self.train_step)。
- 标识模型是否处于训练过程中的 placeholder (self.is_training)。

由于后续在框架的基础上搭建的模型可能会添加额外需要保存的指针，所以我们应该把定义 self.tf_collections 的过程单抽象出来：

```
01 def define_tf_collections(self):
02     self.tf_collections = [
03         "tfx", "tfy", "output", "prob output",
04         "loss", "train step", "is training"
05     ]
```

那么在将这些指针保存下来之后，我们应该如何复用它们呢？TensorFlow 中同样有相应的函数能够帮我们取出保存下来的指针：

```

01 for tensor in self.tf_collections:
02     # 利用 tf.get_collection 来将 tensor 这个名字对应的指针提取出来
03     # 需要特别注意的是, 这里的 target 将会是一个列表 (list)
04     target = tf.get_collection(tensor)
05     if target is None:
06         continue
07     # 规定该指针只能有一个, 否则就是保存或复用指针时出了问题
08     assert len(target) == 1, "{} available '{}' found".format(len(target), tensor)
09     # 利用 setattr 方法, 将指针赋给相应的属性
10     setattr(self, tensor, target[0])

```

其中最后一行用到的 `setattr` 是 `set attribute` 的缩写, 它可以视为 `getattr` 的对偶方法。具体而言, `getattr` 是将类或实例的属性提取出来, `setattr` 是将某个值赋给类或实例的属性。同时需要说明的是, 保存指针时用到的 `tf.add_to_collection` 方法, 其实是会将相应的指针存进一个列表中。具体而言, 如果我们执行

```

for i in range(10):
    tf.add_to_collection(tensor, tar)

```

的话, 那么

```
tf.get_collection(tensor)
```

返回的就将是一个含有 10 个 `tar` 的列表。因此我们在上述代码第 3 行的注释处才说, `target = tf.get_collection(tensor)` 处的 `target` 其实是一个列表。

以上, 我们就完成了 TensorFlow 模型各种基本元素的保存。不过要注意, 我们在搭建模型时会用到各种模型超参数, 所以为了完整地复用模型, 我们必须把这些超参数也保存下来。而正如前文所说, 由于我们把超参数都放在了字典中来管理, 所以只要我们把这两个字典保存下来, 后续就能复现出整个模型。同样的, 由于后续在框架的基础上搭建的模型可能会添加额外需要保存的因素, 所以我们应该把这些需要保存的东西统一放在一个属性中进行管理, 并把这个属性的定义过程单独抽象出来:

```

01 # 我们把所有需要保存的属性都存储在 self.py_collections 中
02 def define_py_collections(self):
03     self.py_collections = [
04         "name", "n_class",
05         "model_param_settings", "model_structure_settings"
06     ]

```

至此, 其实我们已经完成 TensorFlow 模型的保存与复用了。接下来要做的, 是把这些方法进行一个整合。首先来看如何保存与复用超参数相关的属性与指向 `Tensor`、`placeholder` 的指针:

```

01 def save_collections(self, folder):
02     # 先将 self.py_collections 中的内容按名字保存在一个字典中
03     # 然后利用 pickle 这个 Python 标准库将该字典存储到 py.core 中
04     with open(os.path.join(folder, "py.core"), "wb") as file:
05         param_dict = {
06             name: getattr(self, name)

```

```

07         for name in self.py_collections:
08             pickle.dump(param_dict, file)
09         # 调用 self.add_tf_collections 方法, 将必要的指针保存下来
10         self.add_tf_collections()
11
12     def restore_collections(self, folder):
13         # 同样利用 pickle, 把 py.core 中的字典读取出来并进行复用
14         with open(os.path.join(folder, "py.core"), "rb") as file:
15             param_dict = pickle.load(file)
16             for name, value in param_dict.items():
17                 setattr(self, name, value)
18         # 利用之前说过的方法, 复用所有需要的指针
19         for tensor in self.tf_collections:
20             target = tf.get_collection(tensor)
21             assert len(target) == 1, "{} available '{}' found".format(len(target), tensor)
22             setattr(self, tensor, target[0])
23         # 清空储存在 TensorFlow 中的指针
24         self.clear_tf_collections()

```

其中, 最后一行用于清空指针的 `self.clear_tf_collections` 方法的实现如下:

```

01     def clear_tf_collections(self):
02         for key in self.tf_collections:
03             tf.get_collection_ref(key).clear()

```

之所以我们最后一步要清空储存着的指针, 是因为如果此后再调用 `save_collections` 方法的话, 不清空之前存储着的指针就会导致将保存复数的指针。虽然每次复用时都复用最后一个指针是可行的, 但这样做会浪费存储空间, 而且从逻辑上来说也不太对。这也是为什么会要求每个名字对应的指针只能有一个, 否则就是保存或复用指针时出现了问题。

在本节的最后, 我们来看看如何结合本节开头提到的 `tf.train.Saver`, 来实现 TensorFlow 模型的一个完整的、稳定的保存与复用的方法。

```

01     # 保存完整模型的方法
02     # run_id 是该模型的 id, path 是该模型的保存路径
03     # 默认的 path 为 self.model_saving_path
04     def save(self, run_id=0, path=None):
05         if path is None:
06             path = self.model_saving_path
07         # 利用 path 和 run_id 来获取模型保存的文件夹路径
08         folder = os.path.join(path, "{:06}".format(run_id))
09         # 若该路径不存在, 则调用相应函数创建一个文件夹
10         if not os.path.isdir(folder):
11             os.makedirs(folder)
12         print("Saving model")
13         # 把计算图 self.graph 保存下来
14         with self.graph.as_default():
15             # 利用 tf.train.Saver 和 self.save_collections, 完成模型的保存
16             saver = tf.train.Saver()
17             self.save_collections(folder)

```

```

18         saver.save(self. sess, os.path.join(folder, "Model"))
19         print("Model saved to " + folder)
20         return self
21
22     # 复用完整模型的方法，与保存模型的过程几乎一一对应
23     def load(self, run id=None, clear devices=False, path=None):
24         self. model built = True
25         if path is None:
26             path = self. model saving path
27             folder = self. get model name(path, run id)
28             path = os.path.join(folder, "Model")
29             print("Restoring model")
30             # 把之前的 Graph 写入 self. graph 属性
31             with self. graph.as default():
32                 # 利用 TensorFlow 自带的一些函数来完成复用
33                 if self. sess is None:
34                     self. initialize session()
35                     saver = tf.train.import meta graph("{} .meta".format(path), clear devices)
36                     saver.restore(self. sess, os.path.join(folder, "Model"))
37                     self.restore collections(folder)
38                     self. init all settings()
39                     print("Model restored from " + folder)
40         return self

```

其中，模型默认的保存路径 `self.model_saving_path` 是这样定义出来的：

```

01     @property
02     def name(self):
03         return "Base" if self. name is None else self. name
04
05     @property
06     def model saving name(self):
07         return "{} {}".format(self.name, self. name appendix)
08
09     @property
10     def model saving path(self):
11         # os.getcwd 函数中的 cwd 是 “current working directory” 的缩写
12         # 顾名思义，它能够返回调用者所在的路径
13         return os.path.join(os.getcwd(), "_Models", self.model_saving_name)

```

而之所以我们强调了上面定义的 `save` 和 `load` 方法是保存与复用“完整”模型的方法，是因为在保存 `checkpoint` 时，其实并不需要把所有东西都保存下来。具体而言，在 `fit` 的核心循环步骤中曾经指出，我们需要在 `monitor` 告诉我们当前应该保存 `checkpoint` 时将当前模型参数保存下来以便回滚，此时就只需调用 `tf.train.Saver` 的相关函数来把参数存储下来即可：

```

01     def save_checkpoint(self, folder):
02         if not os.path.isdir(folder):
03             os.makedirs(folder)
04         with self._graph.as_default():
05             tf.train.Saver().save(self._sess, os.path.join(folder, "Model"))

```

```

06
07     def restore_checkpoint(self, folder):
08         with self.graph.as_default():
09             tf.train.Saver().restore(self._sess, os.path.join(folder, "Model"))

```

至此，TensorFlow 模型的 save & load 就全部实现完毕了，而事实上，我们也就完成了整个 TensorFlow 模型基本框架的实现。

注意：在本节实现 TensorFlow 模型基本框架的过程中，我们其实在大量的注释里面都把带“test”的属性称为“交叉验证”的相关属性。这主要是因为对于训练的过程而言，从逻辑上来讲是看不到“测试”的相关属性的（除非是特殊的情况，比如 snapshot_ratio 为 0，此时我们不会使用 TrainMonitor，所以把 test 视为测试集也未尝不可）。而且事实上，这种把所有东西都视为交叉验证集的做法，是非常有利于避免引入未知信息从而避免模型陷入过拟合的。

此外，之所以我们让这些属性的名字中含“test”而非含“cv”，主要是为了方便未来（第 7 章）进行拓展。

3.6 朴素神经网络的实现与评估

这一节我们会在上一节实现的基本框架的基础上实现一个较为朴素的神经网络，并在一些人造数据集上简要地评估其性能。为了不让精力分散，我们暂时认为输入的数据都是已经经过数据预处理的（连续型）数据。关于离散型特征的转换方法以及连续型特征的处理手段，我们会在第 5 章和第 6 章中进行相应的说明。

由于上一节实现的基本框架确实非常灵活，所以本节的实现非常简单，代码总长加起来只有几十行，且功能相对而言很完备（包括中间结果的保存、模型整体的保存与复用等），如代码 3.4 所示。

代码 3.4 朴素神经网络的实现：c_BasicNN/NN.py

```

01 # 调用上一节实现的基本框架，以及 NNUtil.py 中的一些辅助工具
02 from _Dist.NeuralNetworks.NNUtil import *
03 from _Dist.NeuralNetworks.Base import Base
04
05 class Basic(Base):
06     # 定义模型的“签名”
07     signature = "Basic"
08
09     """
10     初始化结构
11     name: 模型的名字
12     model_param_settings: 管理“模型超参数”的字典
13     model_structure_settings: 管理“结构超参数”的字典
14     """
15     def __init__(self, *args, **kwargs):

```



```

16     super(Basic, self).__init__(*args, **kwargs)
17     # 除了调用基本框架的初始化方法外，还需要额外地进行一些初始化
18     # 一个是把名称后缀改成 BasicNN，代指“朴素神经网络”
19     self._name_appendix = "BasicNN"
20     # 另一个是定义神经网络特有的两个东西：激活函数和隐藏层的信息
21     self.activations = self.hidden_units = None
22
23     # 定义模型超参数时，只需额外定义激活函数即可
24     # 默认情况下，我们统一使用 ReLU 作为激活函数
25     def init_model_param_settings(self):
26         super(Basic, self).init_model_param_settings()
27         self.activations = self.model_param_settings.get("activations", "relu")
28
29     # 定义结构超参数时，只需额外定义隐藏层信息即可
30     # 默认情况下，我们使用 3 层神经网络，其中
31     # 两个隐藏层的神经元个数都是 256
32     def init_model_structure_settings(self):
33         super(Basic, self).init_model_structure_settings()
34         self.hidden_units = self.model_structure_settings.get(
35             "hidden_units", [256, 256])
36
37     # 搭建一个朴素的神经网络
38     def _build_model(self, net=None):
39         self._model_built = True
40         if net is None:
41             net = self._tfx
42         # 获取原始输入的 shape
43         current_dimension = net.shape[1].value
44         # 如果激活函数是 None，则认为不使用激活函数
45         if self.activations is None:
46             self.activations = [None] * len(self.hidden_units)
47         # 如果是一个字符串，则将它扩充为一个和隐藏层层数等长的 list
48         elif isinstance(self.activations, str):
49             self.activations = [self.activations] * len(self.hidden_units)
50         else:
51             self.activations = self.activations
52         # 利用 for 循环，搭建神经网络中的各个隐藏层
53         for i, n_unit in enumerate(self.hidden_units):
54             net = self._fully_connected_linear(net, [current_dimension, n_unit], i)
55             # 调用 self._build_layer 方法搭建层结构
56             # 其具体实现马上在后文说明
57             net = self._build_layer(i, net)
58             current_dimension = n_unit
59         # 搭建神经网络的输出层并获得原始输出 self._output
60         appendix = "_final_projection"
61         if self.hidden_units:
62             fc_shape = self.hidden_units[-1]
63         else:
64             fc_shape = current_dimension
65         self._output = self._fully_connected_linear(
66             net, [fc_shape, self.n_class], appendix)

```

其中，上述代码 57 行处用到的 `self._build_layer` 方法在现在这个朴素的神经网络中的实现

是比较简单的，它仅仅是根据激活函数来激活当前层的输入而已：

```
01 def build_layer(self, i, net):
02     activation = self.activations[i]
03     # 如果当前激活函数不是 None，则利用 Activations 来激活相应的 net
04     if activation is not None:
05         net = getattr(Activations, activation)(net, "{}{}".format(activation, i))
06     # 否则就意味着不使用任何激活函数，从而直接返回相应的 net
07     return net
```

以上就是朴素神经网络（BasicNN）的所有实现，下面就来看看它的性能究竟如何。为此，笔者实现了三种简单的二维数据集的生成，它们分别如下所述。

- 异或数据集，它认为第一象限和第三象限标签为第一类，第二象限和第四象限标签为第二类，如图 3.13 所示。

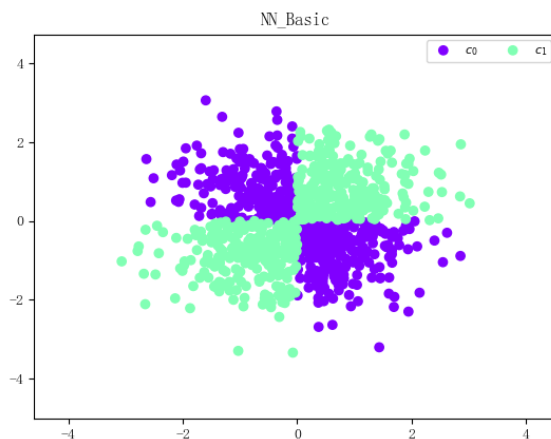


图3.13 异或数据集

- 九宫格数据集，它是异或数据集的“升级版”，认为二维平面的九宫格中，四个角落为第一类，四个边上的为第二类，中间为第三类，如图 3.14 所示。
- 螺旋线数据集，它认为每条螺旋线上的点为一类，如图 3.15 所示。

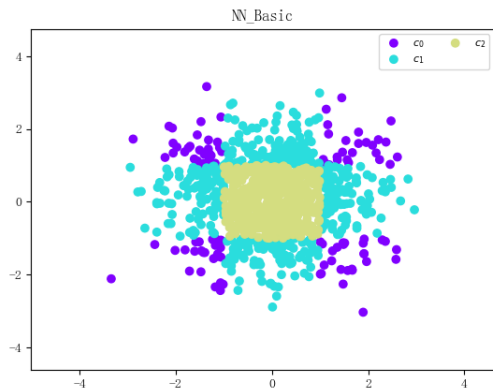


图3.14 九宫格数据集

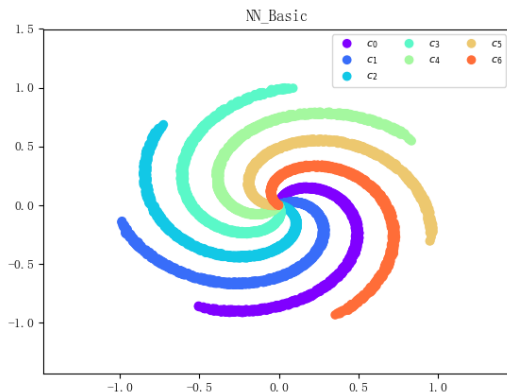


图3.15 螺旋线数据集

这三个数据集都是比较著名的非线性二维数据集，作为测试 BasicNN 而言算是一个不错的选择。为了让大家对模型的表现有更直观的感受，接下来我们不仅会把训练过程中损失函数值的变化曲线（通常称它为“训练曲线”）画出来，而且会把模型的具体分类情况画出来。为了方便讨论与比较，我们统一使用 1000 量级的样本来训练，并采用默认超参数下的神经网络来训练与评估。具体而言，我们会使用 3 层神经网络，其中两个隐藏层的神经元个数都是 256，且激活函数都是 ReLU，优化器统一使用学习速率为 0.001 的 Adam。此时，神经网络在异或数据集上的表现将如图 3.16、图 3.17 所示。

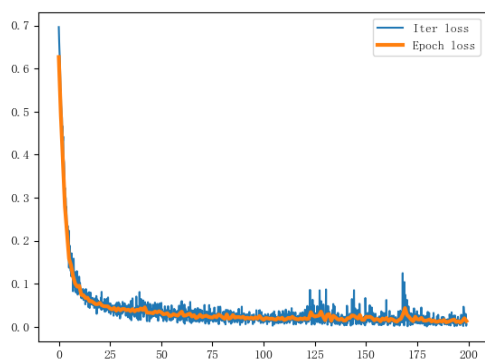


图3.16 异或数据集上的训练曲线

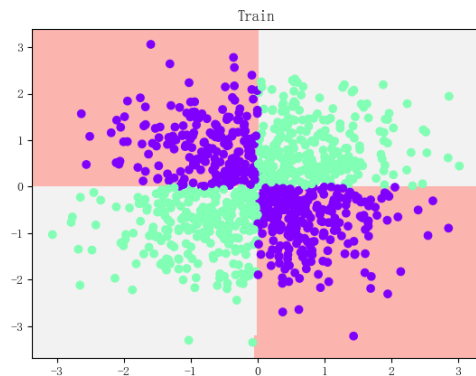


图3.17 异或数据集上的分类情况

注意：接下来将会用到的核心测试代码，可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/c_BasicNN/BasicNN.ipynb。

此时的准确率（acc）是 100%，而且可以看到神经网络学习到了真正的规律——它的分界面几乎正好就是 $x = 0$ 和 $y = 0$ 这两个超平面。

神经网络在九宫格数据集上的表现则如图 3.18、图 3.19 所示。

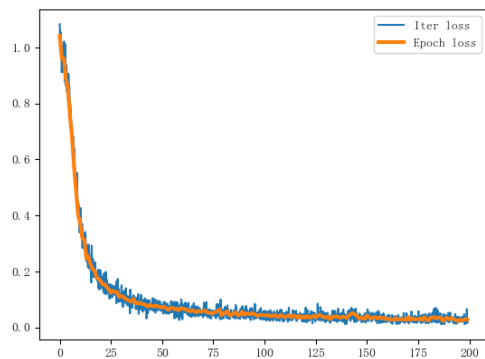


图3.18 九宫格数据集上的训练曲线

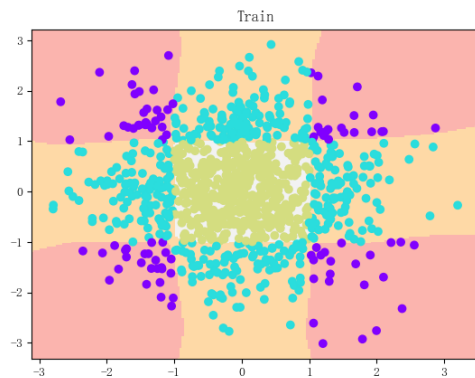


图3.19 九宫格数据集上的分类情况

乍一看似乎表现相当不错，准确率也确实到了 100%，但其实在图像的边缘已经或多或少能够看出，其实神经网络没有像异或数据集那样学到了真实的规律，而只是恰好将训练集拟合正确了而已。事实上，如果我们将视角拉远一点，就会发现它真实的分类情况其实如图 3.20 所示。

也就是说，神经网络学到的分界面不是垂直于坐标轴，而是“越来越偏”的。相比之下，即使我们把视角拉远，异或数据集上的分类情况仍然很好，如图 3.21 所示。

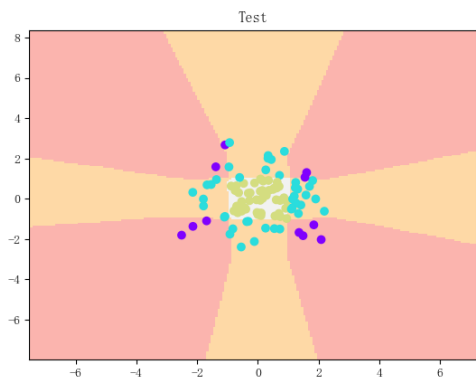


图3.20 九宫格数据集的真实分类情况

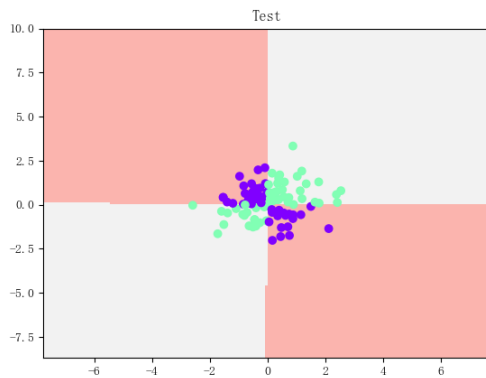


图3.21 异或数据集的真实分类情况

不过九宫格数据集本来就比异或数据集难不少，再加上我们的样本量很少（只是 1000 量级的），所以出现这种情况也不奇怪。

神经网络在螺旋线数据集上的表现则如图 3.22、图 3.23 所示。

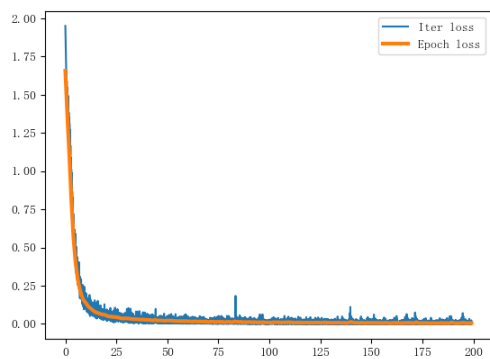


图3.22 螺旋线数据集上的训练曲线

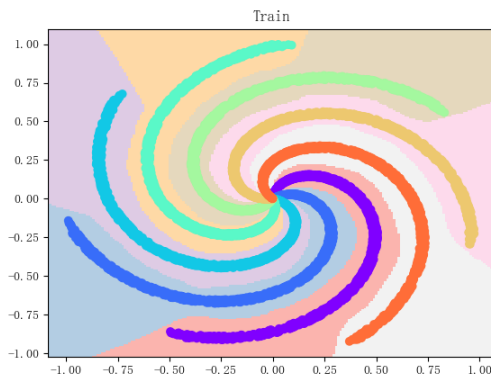


图3.23 螺旋线数据集上的分类情况

由于螺旋线数据集的非线性的感觉更强一点，所以我们从图 3.23 中就能看出，其实神经网络没有真正学习到螺旋的趋势，而仅仅是将训练集的样本都拟合正确了而已。事实上，视角拉远后的分类情况将如图 3.24 所示。

也就是说，神经网络在远处仅仅是用线性函数给二维平面划分出了 7 个区域而已。

注意，我们曾经在 1.1.1 节中给出过这样一个观点：“对神经网络而言，数据的数量往往是最为重要的一环”，现在我们就通过九宫格数据集上的

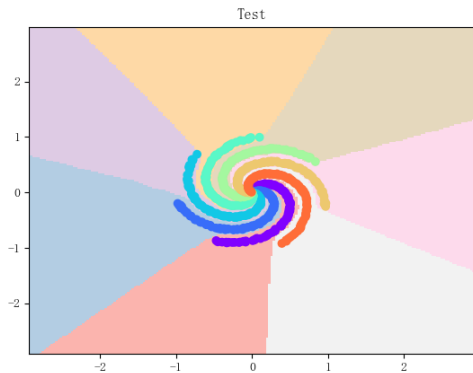


图3.24 螺旋线数据集上的分类情况

实验结果来简要地说明这一点。通过图 3.20 可以看出，当训练数据集样本量在 1000 量级时，神经网络不太能学习到真正的规律。下面我们就用 100 量级和 10 000 量级的训练样本量来进行对比，结果如图 3.25、图 3.26 所示。

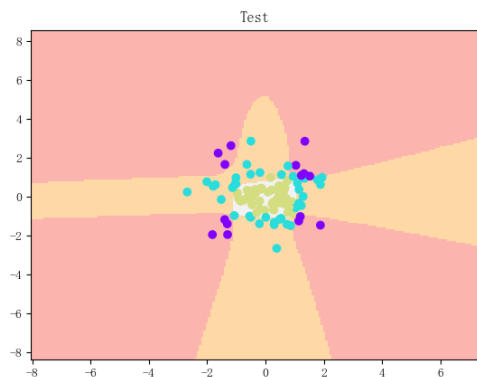


图3.25 100量级的九宫格数据集上的分类情况

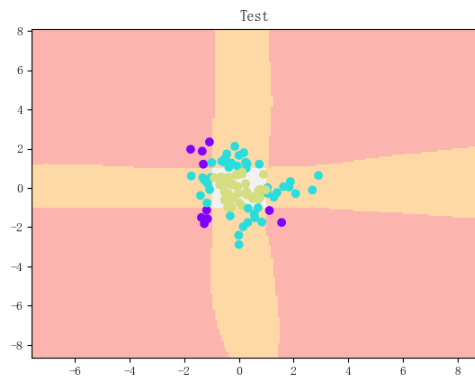


图3.26 10 000量级的九宫格数据集上的分类情况

可以看到，100 量级的结果是一个极度“过拟合”的结果，而 10 000 量级的结果相比之下就显得合理很多。注意，我们采用的模型是完全一致的，这意味着仅通过增加数据量，就能够显著提升神经网络的表现。当然，数据越多则模型表现越好这个规律也不总是正确的，下面就来看一个具体的例子。

之前我们都是非线性的数据集上进行实验，虽然它们确实比较经典，而且能够测试神经网络的拟合能力，但对于神经网络的泛化能力却不能造成太大的挑战。事实上，由于训练集和测试集都是用同一套规则生成出来的，所以此时训练集和测试集的分布是严格一致的。而当我们遇到现实生活中的实际任务时，训练集和测试集的分布往往相差甚远，而这种分布不一致的最常见的原因之一就是数据中有可能有很多噪声。下面我们就构造一个简单的、带噪声的二维线性数据集来测试一下神经网络的泛化能力，该数据集的生成规则如下：

- 在均值为 0、方差为 1 的一维正态分布（即一维标准正态分布）中采样出原始的二维特征向量 \mathbf{x} 。
- 在 \mathbf{x} 的基础上，加上服从均值为 0、标准差为 0.5 的正态分布噪声，得到训练所用的特征向量 $\tilde{\mathbf{x}}$ 。
- 随机初始化权值向量 \mathbf{w} ，并根据 \mathbf{w} 和 \mathbf{x} 的内积是否大于 0 来给相应的带噪声的特征向量 $\tilde{\mathbf{x}}$ 打上标签 y ：

$$y = \begin{cases} 1 & , \text{ if } \mathbf{w}^T \mathbf{x} > 0 \\ 0 & , \text{ if } \mathbf{w}^T \mathbf{x} \leq 0 \end{cases}$$

这就得到了我们的训练集：

$$D = \{(\tilde{\mathbf{x}}_1, y_1), (\tilde{\mathbf{x}}_2, y_2), \dots, (\tilde{\mathbf{x}}_N, y_N)\}$$

注意：在神经网络中，常常还需要对 $y_1 \sim y_N$ 做 OneHot Encoding。

- 至于测试集，则是不带噪声的特征向量与标签的组合。

数据集生成的具体代码如下所示。

```
01 def gen_noisy_linear(size=256, n_dim=2, noise_scale=0.5, test_ratio=1):
02     # 根据样本量 (size) 和特征维度 (n_dim) 来采样原始的特征向量
03     x_train = np.random.randn(size, n_dim)
04     # 加上服从均值为 0、标准差为 noise_scale 的正态分布噪声
05     x_train_noise = x_train + np.random.randn(size, n_dim) * noise_scale
06     # 生成不带噪声的测试集，样本量为训练样本的 test_ratio 倍
07     x_test = np.random.randn(int(size*test_ratio), n_dim)
08     # 随机初始化权重向量 w
09     w = np.random.randn(n_dim, 1)
10     # 根据内积是否大于 0 来打标签
11     y_train = (x_train[..., idx].dot(w) > 0).astype(np.int8).ravel()
12     y_test = (x_test[..., idx].dot(w) > 0).astype(np.int8).ravel()
13     return (x_train_noise, y_train), (x_test, y_test)
```

这样生成出来的、带噪声的（100 量级的）原始数据集将如图 3.27 所示。

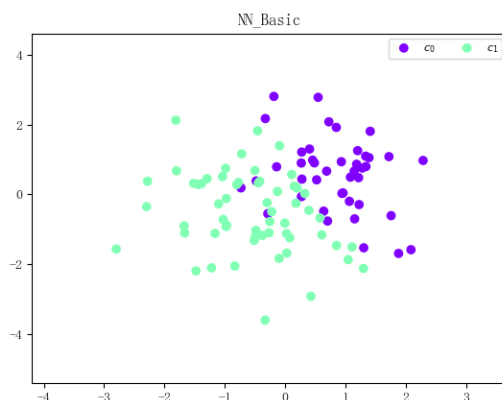


图3.27 带噪声的二维线性数据集

神经网络在这个数据集上的表现将如图 3.28 和图 3.29 所示。

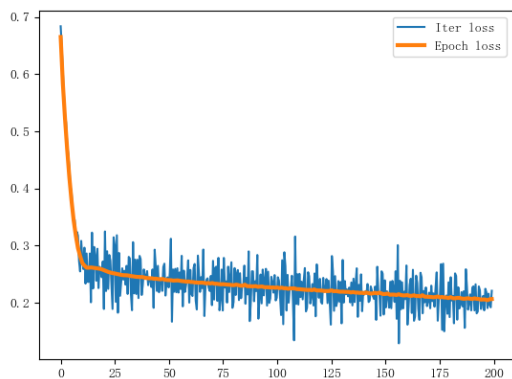


图3.28 带噪声的二维线性数据集上神经网络的训练曲线

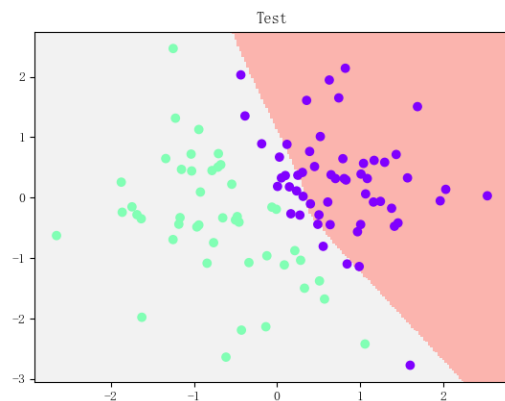


图3.29 带噪声的二维线性数据集上神经网络的分类情况

从图 3.29 中可以看到,虽然线性数据集本身的内在规律非常简单,但由于我们在训练集中加了噪声,再加上神经网络的拟合能力非常强(即使现在我们只实现了朴素的版本,但 **Universal approximation theorem** 仍然是成立的),导致我们过度地拟合了训练集中的噪声,以至于忽略了本质的、简单的规律而找到了一个错误的规律。也正因此,盲目地增加训练样本量很有可能不仅不能提升模型表现,反而让模型越来越趋于过拟合。事实上,1000 量级的训练集上的表现和 10 000 量级的训练集上的表现将分别如图 3.30 和图 3.31 所示。

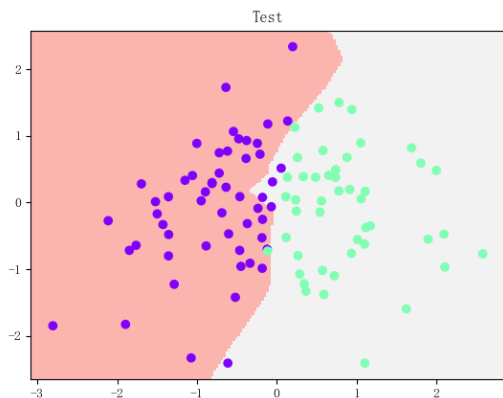


图 3.30 1000量级的噪声二维线性数据集上神经网络的分类情况

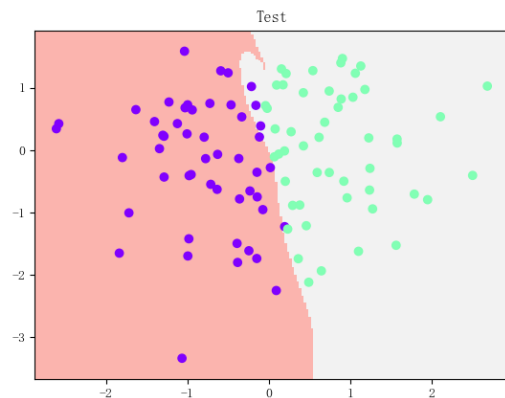


图3.31 10 000量级的噪声二维线性数据集上神经网络的分类情况

可以看到,1000 量级的分界面虽然还算是线性的,但是“崎岖不平”;至于 10 000 量级的分界面,就已经是一个非线性的分界面了。它们的这种表现,很难说比原来 100 量级的表现要好,甚至还可以说是更差了一些。

那么这是否意味着神经网络的能力就仅限于此了呢?其实不然。如果对第 2 章所介绍的 **LinearSVM** 比较熟悉的话,我们可能马上就能感受到,由于它会尝试找到距离正负样本点最远的超平面,而对数据集加的噪声是正态分布噪声,所以它的这种目标恰好可以将噪声的干扰抵消掉,从而意味着它是非常适合拿来做噪声二维线性数据集分类的模型。事实上,它的表现将如图 3.32 所示。

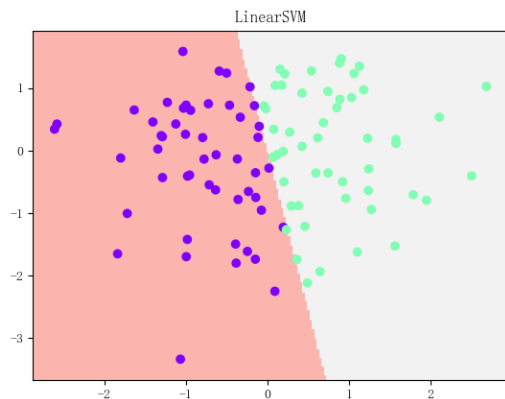


图 3.32 噪声二维数据集上LinearSVM的分类情况

也就是说，它非常完美地学习到了该数据集真正的规律。而 LinearSVM 之所以能够在这个数据集上表现得这么好，很大程度上是因为它的模型很“简单”——它的表达能力远没有神经网络强，但是它却恰好能够适应这种噪声二维数据集。所以如果能让神经网络也有类似的表现的话，比较自然的想法就是让它变“简单”。我们会在第 5 章介绍让神经网络变简单的方法，这里只需感受一下神经网络的局限性并知道我们有相应的解决方案即可。

以上我们就完成了对神经网络的简单评估。需要指出的是，虽然我们用了 4 个数据集来衡量神经网络的性能，但不可否认的是，由于这些数据集都是人造的数据集，所以神经网络在真实任务中的表现确实还不甚明了。对此，我们会在第 7 章用很多真实的数据集来进行实验。之所以不在这里做实验，是因为一旦我们用到了真实的数据集，实现 BasicNN 时所做的假设——认为输入的数据都是已经经过数据预处理的数据——就无法满足了。而至于如何做好封装并实现一个半自动化的、工程化的神经网络框架，则是第 6 章和第 7 章的内容了。

3.7 本章小结

- 神经网络的基本单位是层（Layer），它是一个非常强大的多分类模型。
- 神经网络的基本运算单元是线性映射与激活函数，其中
 - 线性映射中的权值矩阵能将结果从原来的维度空间线性映射到新的维度空间。
 - 线性映射中的偏置量能打破对称性。
 - 激活函数能给神经网络带来非线性性。
- 神经网络通过前向传导算法获取各层的输入与输出，通过反向传播算法来算出各个参数的梯度并利用梯度下降法或其变式来更新参数。
- TensorFlow 模型的组成单元有运算图 Graph、占位符 placeholder 与张量 Tensor，模型的训练则是通过在 Graph 中定义出损失函数后，利用优化器来完成的。
- 将 TensorFlow 模型的各个超参数分类保存在字典中能大大降低保存、复用的难度，而且能够极大地增强模型的灵活性。
- 神经网络的拟合能力非常强，这固然意味着它能适应于各种非线性数据集，但当它遇上带噪声的数据集时，可能会因过度地拟合了噪声而无法学到真实的规律。

第 4 章

从传统算法走向神经网络

第 3 章我们从头开始叙述了神经网络的一些基本理论与具体实现，不过虽然我们介绍了 Universal Approximation Theorem 并在最后给出了一些经典数据集上神经网络的表现，它可能还是显得过于“黑箱”了：我确实知道神经网络是如何（利用前向传导算法）算出结果的，但这毕竟只是数学上诸多公式的堆砌，仅仅凭借上一章的简介可能仍然无法直观地理解神经网络内部究竟发生了什么。而本章的目的，就是想尝试揭开一部分神经网络内部运作机理的神秘面纱。我们将会通过设计上一章实现的朴素神经网络（BasicNN）中的权值矩阵、偏置量与激活函数，来让神经网络表达出第 2 章介绍过的朴素贝叶斯模型与决策树模型。由于这两个模型的可解释性都很强，在实际任务中也各有各的优势，所以如果它们都能转换成神经网络即如果神经网络能够包容它们的话，可能在某种意义上就能比较直观地认识到神经网络模型的逻辑与强大之处了。而对于第 2 章讲到的支持向量机和 Logistic 回归来说，虽然它们也是非常棒的模型，但由于模型的表达式本身就和神经网络一致，区别只在于优化目标不同，所以我们就无须额外地说明如何将它们转换成神经网络了。

本章主要涉及的知识点有：

- 使用神经网络来表达朴素贝叶斯模型
- 使用神经网络来表达决策树模型
- 模型转换的意义与不足之处

4.1 朴素贝叶斯的线性形式

在第 2 章的 2.1.3 节的最后曾经说过，对于任意一个数据集，我们总可以通过某些手段来运用算法 2.1 并搭建出相应的朴素贝叶斯模型。换句话说，我们可以认为

$$G(\mathbf{x}; \theta) = \arg \max_y p(y) \prod_{i=1}^n p(x^{(i)}|y; \theta)^{x^{(i)}}$$

这个公式可以代表相当大一部分的朴素贝叶斯模型。我们接下来要做的，就是把该公式对应的朴素贝叶斯模型用神经网络给表达出来。

在第3章介绍神经网络时我们曾经提到过，神经网络的基本组成单元是层结构，每个层结构是由许多基本运算单元组成的。同时在层结构之间，我们使用了线性映射来作为沟通的桥梁，在层结构内部暂时只使用了激活函数来给神经网络提供非线性性。那么在一个极端的场景下，神经网络将会“退化”成这样一个比较简单的模型：

- 只有输入层和输出层，没有隐藏层。
- 输入层、输出层都不使用激活函数（即不进行数据预处理，也不用变换函数来处理原始输出）。

此时，神经网络的输出即为

$$\mathbf{o}_{NN}(\mathbf{x}) = \mathbf{W}_0 + \mathbf{W}\mathbf{x}$$

注意：可以看到，神经网络此时其实可以视为第2章介绍过的感知机的“向量化版本”：感知机输出的是一个数（或说是一个标量），而神经网络这里输出的通常是一个 K 维向量，其中 K 是分类问题中的类别数目。

可以看出这是一个线性模型，这就说明神经网络是包容线性模型的。所以如果说朴素贝叶斯的本质其实是线性模型的话，那么就意味着神经网络是包容朴素贝叶斯的。但是不难看出，线性模型中涉及很多连加，而朴素贝叶斯的公式里面有很多连乘，这应该如何协调呢？注意我们在2.1.5节中曾经说过，运用对数函数 \log 的话就能把连乘式转为连加式，所以 \log 函数正是将朴素贝叶斯转化为线性模型的关键所在。事实上，由于 \log 函数的图像如图4.1所示，且其导数公式为

$$\frac{\partial \log x}{\partial x} = \frac{1}{x}$$

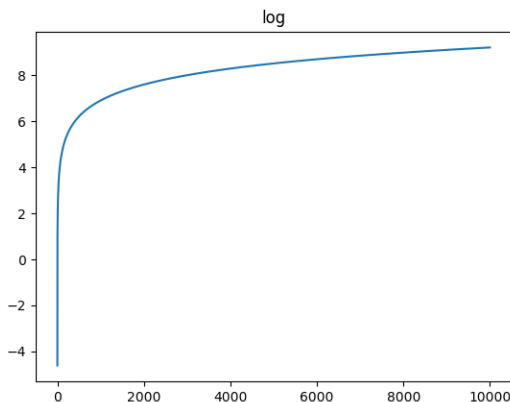


图4.1 \log 函数的函数图像

所以 \log 函数在其定义域 ($x > 0$) 上是单调递增的, 这意味着它不会影响 $\arg\max$ 函数的结果, 即

$$\arg \max_i n_i \equiv \arg \max_i \log n_i, \quad \text{if } n_i > 0 (\forall i)$$

从而朴素贝叶斯模型公式就可以转化为

$$G(\mathbf{x}; \theta) = \arg \max_y \log p(y) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y; \theta)$$

这就是朴素贝叶斯的线性形式。从直观上来说, 此时只需令

$$\mathbf{W}_0 \leftarrow \log p(y), \quad \mathbf{W} \leftarrow \log p(x^{(i)}|y; \theta)$$

那么就相当于做到了朴素贝叶斯向神经网络的转换:

$$G_{NN}(\mathbf{x}) \leftarrow G(\mathbf{x}; \theta)$$

而事实上也正是如此。具体而言, 假设现在有一个 K 类问题, 那么就只需令

$$\mathbf{W}_0 = \begin{bmatrix} \log p(y = c_1) \\ \log p(y = c_2) \\ \vdots \\ \log p(y = c_K) \end{bmatrix}$$

以及 (为了简洁, 省去 θ 不写)

$$\mathbf{W} = \begin{bmatrix} \log p(x^{(1)}|y = c_1) & \log p(x^{(2)}|y = c_1) & \dots & \log p(x^{(n)}|y = c_1) \\ \log p(x^{(1)}|y = c_2) & \log p(x^{(2)}|y = c_2) & \dots & \log p(x^{(n)}|y = c_2) \\ \vdots & \vdots & \ddots & \vdots \\ \log p(x^{(1)}|y = c_K) & \log p(x^{(2)}|y = c_K) & \dots & \log p(x^{(n)}|y = c_K) \end{bmatrix}$$

即可。注意到 $\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^T$, 从而

$$\mathbf{o}_{NN}(\mathbf{x}) = \mathbf{W}_0 + \mathbf{W}\mathbf{x} = \begin{bmatrix} \log p(y = c_1) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_1) \\ \log p(y = c_2) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_2) \\ \vdots \\ \log p(y = c_K) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_K) \end{bmatrix}$$

由于神经网络模型的最终表达式为

$$\begin{aligned} G_{NN}(\mathbf{x}) &= \arg \max_k \mathbf{o}_{NN}(\mathbf{x})_k \\ &= \arg \max_k \log p(y = c_k) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_k) \end{aligned}$$

且朴素贝叶斯模型的表达式也可以写成

$$\begin{aligned}
 G(x) &= \arg \max_y \log p(y) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y) \\
 &= \arg \max_k \log p(y = c_k) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_k)
 \end{aligned}$$

于是从数学上我们也证明了神经网络确实是能表达出朴素贝叶斯模型的。可以看到，这里面并没有特别复杂的地方，只要在第一步能够想到利用 \log 函数并得到朴素贝叶斯模型的线性形式的话，后面的步骤就基本水到渠成了。

以上就是理论上的说明，下面我们来看看具体应该怎样去实现这种“传统算法到神经网络”的转换。由上述的公式推导可以看出，这个转换过程的本质其实只在于本章开头就说到的一点：设计神经网络中的权值矩阵、偏置量与激活函数，所以为了合理复用代码，我们需要先写一个抽象的“转换框架”。无论是本节所介绍的朴素贝叶斯神经网络还是下一节将介绍的决策树神经网络，相应的实现都会在这个“转换框架”上拓展，如代码 4.1 所示。

代码 4.1 转换框架的实现：d_Traditional2NN/Toolbox.py

```

01 class TransformationBase(Basic):
02     """
03     初始化结构
04     *args、**kwargs: 朴素神经网络 BasicNN 的各种参数
05     transform ws: 进行转换所需的权值矩阵的设置
06     transform bs: 进行转换所需的偏置量的设置
07     """
08     def __init__(self, *args, **kwargs):
09         super(TransformationBase, self).__init__(*args, **kwargs)
10         self.transform_ws = self.transform_bs = None
11
12     # 具体进行转换的函数，留待具体的转换算法来定义
13     def transform(self):
14         pass
15
16     # 搭建模型时，先进行转换，再进行搭建
17     def build_model(self, net=None):
18         self.transform()
19         super(TransformationBase, self).build_model(net)
20
21     # 将 self.transform_ws 中的权值矩阵赋给神经网络中的各个权值矩阵
22     def feed_weights(self):
23         for i, w in enumerate(self.transform_ws):
24             if w is not None:
25                 self.sess.run(self.ws[i].assign(w))
26
27     # 将 self.transform_bs 中的偏置量赋给神经网络中的各个偏置量
28     def feed_biases(self):
29         for i, b in enumerate(self.transform_bs):
30             if b is not None:

```

```

31         self.ssess.run(self.bs[i].assign(b))
32
33     # 初始化模型时,除了原基本框架所需要的初始化以外
34     # 还要做一些额外的和模型转换相关的事情
35     def initialize(self):
36         super(TransformationBase, self).initialize()
37         # 将各权值矩阵、偏置量初始化为转换所需的值
38         self.feed_weights()
39         self.feed_biases()
40         # 将神经网络的初始表现输出
41         # 以此来佐证传统算法到神经网络的转换算法的正确性
42         # self.get_all_data方法能利用数据生成器来获取所有训练、测试样本
43         # 由于其实现与本书主题无关,而且也不需要技术性,所以从略
44         x, y, x_test, y_test = self.get_all_data()
45         print("\n".join(["=" * 60, "Initial performance", "-" * 60]))
46         self.evaluate(x, y, x_test, y_test)
47         print("-" * 60)

```

可以看到,由于基本框架(Base)搭建得足够完善,BasicNN实现的灵活性也很强,所以这里这个转换框架的实现非常简单。注意,我们在上述代码的第46行处用了一个叫self.evaluate的方法,它是定义在基本框架Base中的、用于输出模型当前表现的方法,其具体实现代码如下:

```

01     # x,y; x cv,y cv; x test,y test 分别代表训练、交叉验证与测试数据集
02     def evaluate(self, x=None, y=None, x_cv=None, y_cv=None,
03                 x_test=None, y_test=None, metric=None):
04
05         # 如果提供了特定的指标(metric),就使用特定的metric
06         if isinstance(metric, str):
07             metric_name = metric, getattr(Metrics, metric)
08             # 否则,就使用模型超参数中的metric
09             else:
10                 metric_name, metric = self.metric_name, self.metric
11             # 如果提供了相应的数据集,就用self.predict获取相应的预测值
12             pred = self.predict(x) if x is not None else None
13             cv_pred = self.predict(x_cv) if x_cv is not None else None
14             test_pred = self.predict(x_test) if x_test is not None else None
15             # 利用metric获取相应的模型评估指标,该指标能反映模型的表现
16             train_metric = None if y is None else metric(y, pred)
17             cv_metric = None if y_cv is None else metric(y_cv, cv_pred)
18             test_metric = None if y_test is None else metric(y_test, test_pred)
19             # 利用self.print_metrics方法进行输出
20             self.print_metrics(metric_name, train_metric, cv_metric, test_metric)
21             # 返回各个指标
22             return train_metric, cv_metric, test_metric
23
24     @staticmethod
25     def print_metrics(
26         metric_name, train_metric=None,
27         cv_metric=None, test_metric=None, only_return=False

```

```

28 ):
29     # 利用一些字符串技巧, 进行规整的输出
30     print("{} - Train : {}   CV : {}   Test : {}".format(
31         metric name,
32         "None" if train metric is None else "{:8.6}".format(train metric),
33         "None" if cv metric is None else "{:8.6}".format(cv metric),
34         "None" if test metric is None else "{:8.6}".format(test metric)
35     ))

```

至此, 虽说从神经网络的角度来看, 相关的准备已经充分, 不过我们毕竟是要用神经网络来表达朴素贝叶斯, 所以还需要对朴素贝叶斯的相应实现有足够的了解。在第 2 章中我们曾给出过 `sklearn` 中朴素贝叶斯模型的调用方法:

```

01 from sklearn.naive bayes import MultinomialNB
02
03 nb = MultinomialNB()
04 nb.fit(x_train_one_hot, y_train)

```

注意, 朴素贝叶斯转神经网络的核心在于将朴素贝叶斯从原始形式转化为线性形式, 所以我们关心的其实是各个对数先验概率 $\log p(y)$ 以及特征向量中各个维度的特征的对数条件概率 $\log p(x^{(1)}|y) \sim \log p(x^{(n)}|y)$ 。而幸运的是, `sklearn` 里面的朴素贝叶斯模型恰好就有两个属性, 它们分别对应着这两种概率。具体而言, 假设 `nb` 代指 `sklearn` 中的朴素贝叶斯模型, 那么就有 (假设是 K 分类问题):

- `nb.class_log_prior` 是各个对数先验概率, 它是一个 K 维向量, 对应着偏置量。
- `nb.feature_log_prob` 是各个对数条件概率, 它是一个 $K \times n$ 维的矩阵, 对应着权值矩阵。

利用好这两个属性, 相应的实现就很简单了, 如代码 4.2 所示。

代码 4.2 Naïve Bayes→NN 的实现: `d_Traditional2NN/Toolbox.py`

```

01 class NB2NN(TransformationBase):
02     def __init__(self, *args, **kwargs):
03         super(NB2NN, self).__init__(*args, **kwargs)
04         # 定义名字后缀为 NaïveBayes
05         self.name.append("NaïveBayes")
06         # 由于是线性模型, 所以默认不使用激活函数
07         self.model_param_settings.setdefault("activations", None)
08
09     def transform(self):
10         # 由于是线性模型, 所以默认不使用隐藏层
11         self.hidden_units = []
12         x, y, x_test, y_test = self.get_all_data()
13         nb = MultinomialNB()
14         nb.fit(x, y)
15         # 调用 self._print_model_performance 方法能输出 model 的表现
16         # 通过对比该表现与神经网络的初始表现, 就能佐证转换算法的正确性
17         # 同样, 该方法与本书主题关系不大, 故从略

```

```

18     self.print_model_performance(nb, "Naive Bayes", x, y, x_test, y_test)
19     # 利用上文提到的两个属性，对权值矩阵与偏置量做初始化设置
20     # 注意在 3.5.2 的最后曾说过，实际实现时的权值矩阵事实上是
21     # 推导时的权值矩阵的转置，所以这里对权值矩阵做初始化设置时，
22     # 需要相应地做一个转置
23     self.transform_ws = [nb.feature_log_prob_.T]
24     self._transform_bs = [nb.class_log_prior_]

```

下面我们就来看看具体的效果。为了和第2章相呼应，我们仍然使用蘑菇数据集来做实验，初始结果如图4.2所示。

可以看到，至少从结果上来说，我们可以自信地认为转换算法是正确的。

以上，我们就证明了无论是从原理还是实现上来说，用神经网络来表达朴素贝叶斯都是一件比较容易的事，这得益于朴素贝叶斯的本质是一个线性模型，从而可通过神经网络来表达出朴素贝叶斯的线性形式，我们就能完成这两者之间的转换。

```

=====
Naive Bayes performance
=====
acc - Train : 0.955167  CV : 0.95339
=====
Initial performance
=====
acc - Train : 0.955167  CV : 0.95339  Test : None
=====

```

图4.2 Naïve Bayes→NN算法的初始结果

4.2 决策树生成算法的本质

在第2章叙述决策树算法时，我们曾经给决策树的生成归纳出了这样两条核心性质：

- 决策树的生成是不断地挑选出特征并对特征向量空间进行划分的过程。当空间被划分到信息量只剩很少时，我们就给这个子空间打上一个标签。于是当一个新的特征向量进入模型时，我们只需要看它属于哪个子空间，然后把相应的标签输出即可。
- 被划分出来的子空间彼此之间是互不相交的，即对于任意一个新的特征向量而言，它属于且仅属于其中某一个子空间。

同时，在第2章介绍SVM时我们曾经提到过，决策树的生成还需要多两个限制。具体而言，决策树一般只会用超平面来划分特征空间，而且这些超平面一般会垂直于坐标轴。所以决策树生成算法的本质从直观上来说，会有如下三个要点：

- 决策树中的每个中间节点都对对应着一个（垂直于坐标轴的）超平面。
- 决策树中的每条决策路径都对对应着一个超平面的组合，该组合对应着一个子空间。
- 决策树中的每个叶子节点都对对应着这样一个子空间，这些子空间构成了原特征向量空间的划分。

所以如果我们想用神经网络来表达出决策树的话，需要时刻记住决策树生成算法的这三个本质属性。而事实上在我们设计出来的神经网络中，会有如下三条对应的核心性质：

- 第1个隐藏层能表达出决策树的中间节点所对应的超平面。
- 第2个隐藏层能够表达出各个决策路径。

- 输出层能够表达出各个叶节点。

下面我们就来看看在实际操作层面上，应该如何进行具体的构造以使得神经网络满足这样三条核心性质。不失一般性，我们统一用“X 数据集”来进行相应的说明。X 数据集是一个二维数据集，它的生成规则如下：

- 在二维平面上画两条直线，这两条直线会把二维平面分成四个部分。
- 一共分两类，且位置相对的两个部分为同一类。

在这种生成规则下，1000 量级的 X 数据集将如图 4.3 所示。

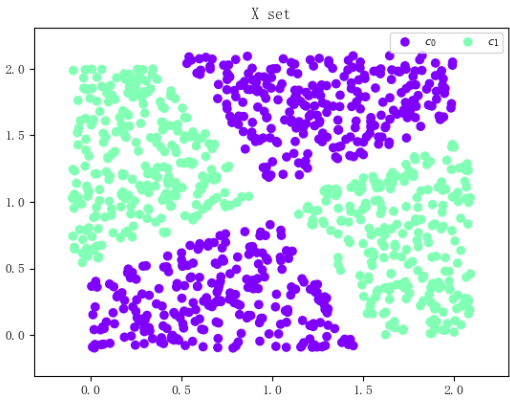


图4.3 1000量级的X数据集

其中，偏右上和偏左下的样本点处于相对的位置，所以它们属于同一类（比如，认为这些样本点的标签 $y = 0$ ）；偏左上和偏右下的样本点也处于相对的位置，所以它们也属于同一类（比如，认为这些样本点的标签 $y = 1$ ）。同时不难看出，图 4.3 所对应的数据集其实是采用了

$$\begin{cases} 0.5 \cdot (x_1 - 1) - x_2 + 1 = 0 \\ -2 \cdot (x_1 - 1) - x_2 + 1 = 0 \end{cases}$$

这两条直线的 X 数据集，其中 x_1 、 x_2 分别代表着图 4.3 中的横、纵坐标。由 2.3.1 节给出的定义可知，对于二维平面而言，这两条直线其实就是超平面。

为了方便，我们先暂时去掉决策树中“超平面一般会垂直于坐标轴”这个限制，从而图 4.3 所示的 X 数据集所对应的“最好的决策树”将如图 4.4 所示。

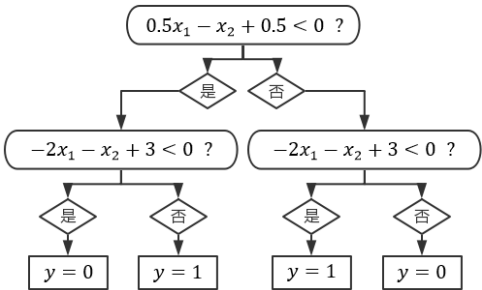


图4.4 图4.3所示的X数据集所对应的“最好的”决策树模型

接下来的讨论都将基于图 4.4 展开。需要指出的是，我们会在 4.2.4 节中把“超平面一般会垂直于坐标轴”这个限制再加回来，从而体现出决策树的局限性。

注意：使用神经网络来表达决策树并非笔者独创，前人已经有不少相关讨论，其中的“开山之作”大抵是 *Entropy Nets - From Decision Trees to Neural Networks* 这一篇。不过根据笔者的调研来看，相关的论文叙述得都有些不甚清晰，给出了具体算法以及相应实现的则几乎没有。所以笔者就综合了各篇论文的共通点以及自己对这个问题的理解写出了这一章的内容，希望它能帮助大家更轻松、全面地掌握相应知识点。

4.2.1 第 1 隐藏层→决策超平面

首先我们来看看如何具体构造出第一个核心性质，即如何用第 1 个隐藏层中的神经元来表达出决策树中间节点所对应的超平面。换句话说，我们的目标是让第 1 隐藏层中的各个神经元和决策树中的各个超平面形成一一对应的关系。具体到我们要讨论的问题，我们希望在第 1 隐藏层中构造出两个神经元 u_1 、 u_2 ，它们能够分别表达出 Π_1 、 Π_2 这两个超平面，其中

$$\begin{cases} \Pi_1 &: 0.5x_1 - x_2 + 0.5 = 0 \\ \Pi_2 &: -2x_1 - x_2 + 3 = 0 \end{cases}$$

这一步的构造其实相当直观：我们只需根据各个超平面的系数和偏置量，定出对应神经元相应的权重和偏置量即可。比如，超平面 Π_1 的系数和偏置量分别为

$$a_1 = 0.5, a_2 = -1, \quad b = 0.5$$

那么对于神经元 u_1 而言，它的权重和偏置量其实就是

$$W_{11}^{(1)} = a_1 = 0.5, W_{21}^{(1)} = a_2 = -1, \quad W_{01}^{(1)} = b = 0.5$$

示意图则如图 4.5 所示。

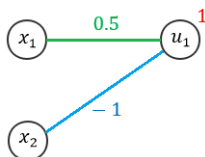


图4.5 用于表达 Π_1 的神经元 u_1 所对应的权值与偏置量

这意味着神经元 u_1 的输入值其实就是

$$h_1^{(1)} = 0.5x_1 - x_2 + 1$$

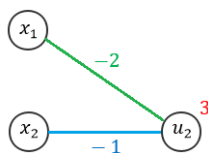
类似的，超平面 Π_2 的系数和偏置量分别为

$$a_1 = -2, a_2 = -1, \quad b = 3$$

从而对于神经元 u_2 而言，它的权重和偏置量其实就是

$$W_{12}^{(1)} = a_1 = -2, W_{22}^{(1)} = a_2 = -1, \quad W_{02}^{(1)} = b = 3$$

示意图则如图 4.6 所示。

图4.6 用于表达 Π_2 的神经元 u_2 所对应的权值与偏置量

这意味着神经元 u_2 的输入值其实就是

$$h_2^{(1)} = -2x_1 - x_2 + 3$$

综上所述，第1隐藏层相对应的权值矩阵与偏置量的设置就应如图4.7所示。

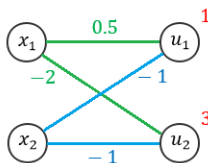


图4.7 用于表达决策树中间节点对应的超平面的第1隐藏层

以上我们就完成了第1隐藏层的权值矩阵与偏置量的设置。而对于神经网络的基本运算单元而言，除了权值矩阵与偏置量所带来的线性映射以外，我们还需要激活函数带来非线性。具体到决策树转神经网络的算法的话，第1隐藏层所对应的非线性其实就是“表达出超平面”的能力。由前文的讨论可知，此时神经元 u_1 、 u_2 所接收的输入分别是

$$h_1^{(1)} = 0.5x_1 - x_2 + 1, \quad h_2^{(1)} = -2x_1 - x_2 + 3$$

那么不难看出，只需再将基本运算单元中的激活函数设为2.3.1节中提到过的符号函数 sign ，就能让第一个隐藏层表达出对应的超平面。具体而言，由于 sign 的公式是：

$$\text{sign}(x) = \begin{cases} -1 & , \quad x < 0 \\ 1 & , \quad x \geq 0 \end{cases}$$

所以可以发现，此时我们的神经网络有这样一个特性：当特征向量在某个中间节点往左走——比如它在图4.4所示的决策树中的根节点处向左走——时，由于此时它满足

$$0.5x_1 - x_2 + 0.5 < 0$$

于是对应的神经元 u_1 的输入值 $h_1^{(1)}$ 将小于0，于是经过激活函数 sign 后的最终（非线性）输出值将会是-1。同理，如果它在某个中间节点往右走，那么对应神经元的最终输出值将会是1。从直观上来说，这种对应关系就可以使第一个隐藏层拥有表达对应的超平面的能力。在下一节的相关讨论中我们将会更清晰地认识到这一点，这里暂时按下不表。

4.2.2 第2隐藏层→决策路径

接下来看看如何构造出第二个核心性质，即如何用第2隐藏层来表达出决策树的决策路径。与上一节中试图让第1隐藏层中的各个神经元和决策树中的各个超平面形成一一对应的关系类似，在这一节中我们将试图让第2隐藏层中的各个神经元和决策树中的各个决策路径形成一一对应的关系。具体而言，由图4.4可知，我们有4条决策路径：

$$\begin{cases} l_1 : 0.5x_1 - x_2 + 0.5 < 0 \text{ and } -2x_1 - x_2 + 3 < 0 \\ l_2 : 0.5x_1 - x_2 + 0.5 < 0 \text{ and } -2x_1 - x_2 + 3 \geq 0 \\ l_3 : 0.5x_1 - x_2 + 0.5 \geq 0 \text{ and } -2x_1 - x_2 + 3 < 0 \\ l_4 : 0.5x_1 - x_2 + 0.5 \geq 0 \text{ and } -2x_1 - x_2 + 3 \geq 0 \end{cases}$$

我们想做的，就是在第2隐藏层中用4个神经元 $v_1 \sim v_4$ 来分别表达出这4条决策路径。注意到在上一节末我们曾经说过，第1隐藏层的两个神经元 u_1 、 u_2 有这样的性质：

- 当 $0.5x_1 - x_2 + 0.5 < 0$ 、 ≥ 0 时， u_1 的输出值分别为-1、1。
- 当 $-2x_1 - x_2 + 3 < 0$ 、 ≥ 0 时， u_2 的输出值分别为-1、1。

为了简便，我们用 $o(u)$ 来代指神经元 u 的输出，从而上述4条决策路径就满足：

$$\begin{cases} l_1 : o(u_1) = -1 & o(u_2) = -1 \\ l_2 : o(u_1) = -1 & o(u_2) = 1 \\ l_3 : o(u_1) = 1 & o(u_2) = -1 \\ l_4 : o(u_1) = 1 & o(u_2) = 1 \end{cases}$$

在得到每条决策路径所对应的这些输出值后，用第2隐藏层来表达决策路径的方法就很直观了：对于权值矩阵而言，只需让相应神经元的权值符号和相应输出值的符号一致，并把权值大小设为决策路径上神经元个数的倒数，对于偏置量则统一设置为0。具体而言，假设某条决策路径 l 上有 n 个超平面 $\Pi_1 \sim \Pi_n$ ，这 n 个超平面在第1隐藏层中对应的神经元为 $u_1 \sim u_n$ ，那么这条决策路径在第2隐藏层中所对应的神经元 v 和 $u_1 \sim u_n$ 之间的权值设置就应该是：

$$W_{u_i v}^{(2)} = o(u_i) \cdot \frac{1}{n}$$

且对于第1隐藏层中的其他神经元而言， v 和它们之间的权值都设置为0。此时不难发现，如果某个特征向量 x 属于决策路径 l 的话，由于此时神经元 $u_1 \sim u_n$ 的输出为 $o(u_1) \sim o(u_n)$ ，所以神经元 v 所接收到的输入就是

$$h_v^{(2)} = \sum_{i=1}^n W_{u_i v}^{(2)} \cdot o(u_i) = \sum_{i=1}^n o^2(u_i) \cdot \frac{1}{n} = 1$$

这是因为 $o(u_i) \in \{-1, 1\}$ ，从而总有 $o^2(u_i) = 1$ 。注意我们曾经说过，每个特征向量都会属于且仅属于一条决策路径，所以在第2隐藏层中，将会有且仅有一个神经元接收的输入为1。从直观上来说的话，这个性质就使得第2隐藏层拥有了表达出各决策路径的能力。

具体到我们要讨论的问题中的话，由于所有决策路径上的神经元个数都是两个，所以所有权值大小都是0.5。同时根据决策路径上 u_1 、 u_2 的输出值，我们能直接完成第1隐藏层到第2隐藏层的权值矩阵的设置：

$$\begin{cases} u_1 \rightarrow v_1 \sim v_4 : -0.5 & -0.5 & 0.5 & 0.5 \\ u_2 \rightarrow v_1 \sim v_4 : -0.5 & 0.5 & -0.5 & 0.5 \end{cases}$$

示意图如图4.8所示。

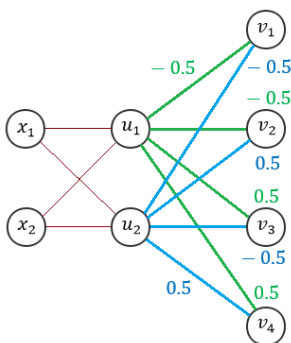


图4.8 用于表达决策树各决策路径的第2隐藏层

4.2.3 输出层→叶节点

最后来看看如何构造出第三个核心性质，即如何构造输出层表达出决策树中的叶节点。只要利用上一节所说的第2隐藏层的性质，这一步其实是非常直观的：由于在第2隐藏层中，只有代表了指定决策路径的神经元会接收到1的输入值，其余神经元接收到的值都会比1小。用数学语言来叙述的话，就是如果假设我们现在有 $n^{(2)}$ 条决策路径 $l_1 \sim l_{n^{(2)}}$ ，它们在第2隐藏层中对应的神经元为 $v_1 \sim v_{n^{(2)}}$ ，则会有

$$\mathbf{x} \in l_i \Rightarrow h(v_i) = 1, \quad h(v_j) = 1 - \epsilon_j \quad (j \neq i)$$

其中， $h(v)$ 代指神经元 v 的输入值，且对任意的 j 而言，都有 $\epsilon_j > 0$ 。

那么在此基础上，我们只需将第2隐藏层的激活函数设置为类似于 OneHot 感觉的激活函数即可。所谓的 OneHot 感觉的激活函数，是指除了对各个输入中的最大值进行保留以外，其余输入值都进行较大幅度的衰减的函数。最极端的情况就是，除了对最大输入值1进行保留以外，其余输入值都衰减为0，即

$$\phi^{(2)}(\mathbf{h}) = \left(0, \dots, h(v_i), \dots, 0\right)^T = \left(0, \dots, 1, \dots, 0\right)^T$$

其中， $\mathbf{x} \in l_i$ 且 $\mathbf{h} = \left(h(v_1), h(v_2), \dots, h(v_{n^{(2)}})\right)^T$ 。

注意：我们曾在3.5.2节中介绍 Activations 类的实现时介绍过上述激活函数的实现，感兴趣的读者可以回头看看当时的实现是否与现在的这个描述相符。

在这种情况下，第2隐藏层中各个神经元的输出就拥有这样一个很强的性质：除了指定决策路径 l_i 对应的神经元的输出 $o(v_i)$ 为1以外，其余神经元输出的都是0。注意，输出层的各个神经元其实就代表着分类问题中的各个类别，所以此时只需要将 v_i 这个神经元连接到决策路径 l_i 所对应的叶节点所属类别的神经元上即可。而连接的权值则可以视情况而定，常见的做法是按照叶节点中各个样本的标签比例来分配权值。具体而言，假设决策路径 l_i 对应的叶节点中有2个 $y=0$ 的样本、8个 $y=1$ 的样本，同时假设输出层中代表着第 k 个类别的神经元为 o_k ，即

$$o_1 \leftrightarrow y = 0, \quad o_2 \leftrightarrow y = 1$$

则相应的权值就可以设置为：

$$v_i \rightarrow o_1, o_2 : 0.2 \quad 0.8$$

而如果 l_i 的叶节点中全是 $y = 0$ 的样本的话，相应的权值就可以设置为：

$$v_i \rightarrow o_1, o_2 : 1 \quad 0$$

偏置量则和第2隐藏层一样，统一设置为0即可。此时不难看出，输出层确实拥有了表达出决策树中的各个叶节点的能力。

具体到我们要讨论的问题，由于规则已经是定好了的，所以每条决策路径对应的叶节点上都只可能有一个类别的样本点。具体而言，由图4.4可直接得知

$$\begin{cases} l_1 : y = 0 \\ l_2 : y = 1 \\ l_3 : y = 1 \\ l_4 : y = 0 \end{cases}$$

从而权值就应该设置为

$$\begin{cases} v_1 \rightarrow o_1, o_2 : 1 \quad 0 \\ v_2 \rightarrow o_1, o_2 : 0 \quad 1 \\ v_3 \rightarrow o_1, o_2 : 0 \quad 1 \\ v_4 \rightarrow o_1, o_2 : 1 \quad 0 \end{cases}$$

示意图则如图4.9所示。

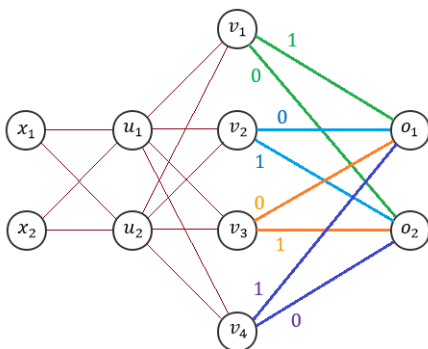


图4.9 用于表达决策树各个叶节点的输出层

4.2.4 具体实现

在前3节中，我们依次证明了构造出拥有本节开头所说的三条核心性质的神经网络是可行的。那么在这一节中，我们将把前三节的内容做一个整合，并介绍如何将完整的构造过程实现出来。与朴素贝叶斯神经网络不同，决策树神经网络的实现是相当复杂的，所以我们需要好好理清逻辑后再看具体的代码。

首先需要解决的就是需要获取哪些以及如何获取决策树中间节点与叶节点的信息这个问题。在第2章曾经说过，scikit-learn的DecisionTreeClassifier会把所有的特征都视为连续型特征，同时它每次只会挑选出一个特征来作为划分的依据。换句话说，scikit-learn的决策树将会是一棵二叉树，且它的中间节点对应的超平面是垂直于坐标轴的超平面的，这种处理方式其实大大简

化了问题的复杂度。具体而言，我们现在只需关心：

- 中间节点对应的特征 $x^{(i)}$ 的特征维数 i 以及相应的阈值 $\tilde{\epsilon}_i$ ，从而该中间节点对应的超平面即为 $x^{(i)} = \tilde{\epsilon}_i$ 。
- 叶节点中各个样本所构成的类别分布。比如一个叶节点中有 2 个第一类别的样本和 8 个第二类别的样本，我们就只需获得[0.2 0.8]这个向量而不必关心这 10 个样本本身。

幸运的是，这些信息在 scikit-learn 的决策树模型中都能调用出来，所以实现的基石就已经打好了。不过，虽说信息是完备的，我们仍然需要设计一种高效的、无冗余、无缺漏的调用方式。合适的调用方式有很多，这里就仅介绍其中一种最为直观的做法：先序遍历（Pre-Order Traversal）。先序遍历采用了递归的思想，其具体遍历手段如下（注意我们只需关心二叉树的情况，因为 scikit-learn 生成的决策树都是二叉树）：

- 在某个节点处，总是先往左子节点搜索，再往右子节点搜索。
- 除非在某条搜索路径上达到了叶节点，否则不会往回搜索。

下面就以图 4.4 所示的二叉（决策）树为例，具体说明一下先序遍历的搜索方式。为了简便，我们使用图 4.10 所示的（等价的）二叉树来进行讨论。

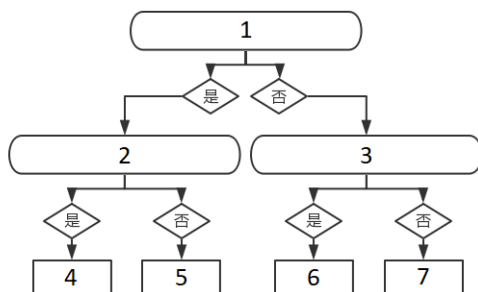


图4.10 先序遍历示例

在图 4.10 所示的二叉树中，先序遍历的搜索顺序如下所示：

- 1 → 2 → 4，发现是叶节点，返回 2 并往右搜索。
- 2 → 5，发现是叶节点，返回 1 并往右搜索。
- 1 → 3 → 6，发现是叶节点，返回 3 并往右搜索。
- 3 → 7，发现是叶节点，然后发现已经将整棵树都搜索完毕了，于是终止搜索。

不难看出，先序遍历的这种遍历方式与我们的最终目的是非常契合的。事实上不难发现，先序遍历每一次“回头”时都意味着它恰好搜索完了一条完整的决策路径，所以我们只需在它“回头”时，把对应的决策路径上的信息进行采集、汇总即可。

具体的信息整合方式我们会在后文进行说明，这里就先来看看如何实现先序遍历。为此，我们需要先对 scikit-learn 的 DecisionTreeClassifier 中的一些属性做一个介绍。假设 tree 是 DecisionTreeClassifier 模型的一个实例，那么它的 tree_ 属性将会包含我们想要的信息。事实上，tree.tree_ 的各个属性分别如下所示。

- **feature**: 储存着各个中间节点对应的特征维数 i 的列表 (list)。
- **threshold**: 储存着各个中间节点对应的阈值 ϵ_i 的列表。
- **value**: 储存着各个中间节点对应的类别分布的列表。
- **children_left**: 储存着各个中间节点的左子节点的下标的列表。
- **children_right**: 储存着各个中间节点的右子节点的下标的列表。

同时, 根节点的下标必定是 0, 即上述 5 个属性所对应的列表的第 1 个元素都必定是根节点的相应信息。那么利用这些性质, 先序遍历的实现就比较直观了, 如代码 4.3 所示。

代码 4.3 先序遍历的实现: d_Traditional2NN/Toolbox.py

```

01 # 导入 sklearn 中辅助我们进行信息提取的文件
02 from sklearn.tree import tree
03
04 # 定义先序遍历的主函数, 其中 tree 代指决策树模型 DecisionTreeClassifier
05 def export_structure(tree):
06     # 调用 tree 的 tree 属性以辅助我们进行信息提取
07     tree = tree.tree
08     # 定义核心的递归函数, 其中
09     # node 代表当前节点 (的下标), depth 代表当前节点的深度
10     def recurse(node, depth):
11         # 获取当前的特征维数
12         feature_dim = tree.feature[node]
13         # tree.TREE_UNDEFINED 是“无”的标识
14         # 从而 feature_dim == tree.TREE_UNDEFINED 意味着 sklearn 没有给
15         # 当前节点分配特征, 这就意味着当前节点是叶节点
16         if feature_dim == tree.TREE_UNDEFINED:
17             # 于是 yield 出相应的信息, 注意我们在标识特征维数的地方设为了 -1
18             # 从而调用者如果发现特征维数是 -1 的话,
19             # 就能知道该节点是叶节点了
20             yield depth, -1, tree.value[node]
21         else:
22             # 否则的话, 先获取当前的阈值
23             threshold = tree.threshold[node]
24             # 然后 yield 出当前的信息
25             yield depth, feature_dim, threshold
26             # 并将左子节点所对应的所有信息以先序遍历的方式 yield 出来
27             yield from recurse(tree.children_left[node], depth + 1)
28             # 继而再 yield 出当前的信息, 提示调用者
29             # “已经遍历完左子节点, 即将遍历右子节点”
30             yield depth, feature_dim, threshold
31             # 最后以先序遍历的方式将右子节点所对应的所有信息 yield 出来
32             yield from recurse(tree.children_right[node], depth + 1)
33
34     # 从根节点开始执行递归 (注意根节点下标必为 0)
35     # 同时将结果转成一个 list 返回
36     return list(recurse(0, 0))

```

接下来就要具体实现转换的过程了。由于该实现确实有比较强的逻辑性, 所以我们把这段

实现分为两步来说明：第一部分是前期的准备，第二部分是信息的整合。首先来看第一部分，这一步的实现还是比较易懂的，如代码 4.4 所示。

代码 4.4 Decision Tree→NN 的实现：d_Traditional2NN/Toolbox.py

```

01 class DT2NN(TransformationBase):
02     def __init__(self, *args, **kwargs):
03         super(DT2NN, self).__init__(*args, **kwargs)
04         # 定义名字后缀为 DTree
05         self._name_appendix = "DTree"
06         # 默认使用 sign 和 one_hot 这两个激活函数以完成等价转换
07         self.model_param_settings.setdefault("activations", ["sign", "one_hot"])
08
09     def _transform(self):
10         # 这一部分与 Naïve Bayes→NN 处的实现是类似的
11         x, y, x_test, y_test = self._get_all_data()
12         tree = DecisionTreeClassifier()
13         tree.fit(x, y)
14         self._print_model_performance(tree, "Decision Tree", x, y, x_test, y_test)
15
16         # 在训练好决策树后，利用 export_structure 提取出所需的信息
17         tree_structure = export_structure(tree)
18         # 正如前文所说，特征维数为-1意味着该节点是叶节点
19         # 所以利用该性质，我们就能统计出叶节点的数量
20         n_leafs = sum([1 if pair[1] == -1 else 0 for pair in tree_structure])
21         # 由二叉树的性质可知，中间节点的数量是叶节点的数量减 1
22         n_internals = n_leafs - 1
23
24         # 将中间节点数量和叶节点数量都 print 出来
25         # 这两个数其实就是第 1 隐藏层和第 2 隐藏层的神经元个数
26         print("Internals : {} ; Leafs : {}".format(n_internals, n_leafs))
27
28         # 初始化偏置量。由于只有第 1 隐藏层需要偏置量，所以只用初始化一个
29         b = np.zeros(n_internals, dtype=np.float32)
30         # 相对应的，权值矩阵则需要初始化三个
31         w1 = np.zeros([x.shape[1], n_internals], dtype=np.float32)
32         w2 = np.zeros([n_internals, n_leafs], dtype=np.float32)
33         w3 = np.zeros([n_leafs, self.n_class], dtype=np.float32)
34         # 定义一些辅助信息整合的变量，它们的实际意义会马上在后文给出
35         node_list = []
36         node_sign_list = []
37         node_id_cursor = leaf_id_cursor = 0
38         # 记录最长决策路径长度的变量
39         max_route_length = 0
40         # 根据中间节点数量和叶节点数量，定出神经网络中
41         # 两个隐藏层的神经元个数
42         self.hidden_units = [n_internals, n_leafs]

```

其中，上述代码的第 35 行到第 37 行处定义了 4 个关键的、整合信息时会用到的变量，了

解并熟记它们的实际意义对于了解接下来的信息整合的实现是非常有必要的。具体而言，这4个变量的内涵分别如下所述。

- **node_list**: 储存着当前决策路径上各个中间节点的下标的列表。
- **node_sign_list**: 储存着当前决策路径上，样本在各个中间节点处是往左走还是往右走的列表。由第2节的讨论可知，当样本在某个中间节点处往左走时，该中间节点对应的神经元的输出将会是-1，从而 **node_sign_list** 中的相应元素也将是-1。
- **node_id_cursor**: 标记当前的中间节点是见过的“第几个”中间节点。
- **leaf_id_cursor**: 标记当前的叶节点是见过的“第几个”叶节点。

只要利用好这些变量，实现第1个核心性质——即使用第1隐藏层中的神经元来表达出各个中间节点对应的超平面——就比较简单了：

```

43     for depth, feat_dim, rs in tree_structure:
44         # 如果当前节点不是叶节点的话，就记录下各种信息
45         if feat_dim != -1:
46             # 如果当前节点深度和 node_list 长度一致
47             # 就意味着正在往左搜索
48             if depth == len(node_list):
49                 # 从而 node_sign_list 就应该 append 一个
50                 # -1 以表示接下来要往左
51                 node_sign_list.append(-1)
52                 # node_list 则应该 append 该节点的各种信息
53                 node_list.append([node_id_cursor, feat_dim, rs])
54                 # 由于 scikit-learn 中所有超平面都形如  $x^{(i)} = \tilde{\epsilon}_i$ ，即
55                 #  $x^{(i)} - \tilde{\epsilon}_i = 0$ ，所以相应的权值都应该设置为 1
56                 w1[feat_dim, node_id_cursor] = 1
57                 # 相应的偏置量则应该设置为  $-\tilde{\epsilon}_i$ 
58                 b[node_id_cursor] = -rs
59                 # 最后，把 node_id_cursor 加 1，表示我们又
60                 # “见过了”一个中间节点
61                 node_id_cursor += 1
62             # 否则，就意味着我们的搜索是从某个叶节点处返回的
63             # 换句话说，这意味着接下来应该往右搜索
64         else:
65             # 从而应该将该节点及之前节点的信息进行保留
66             node_list = node_list[:depth + 1]
67             # 并对 node_sign_list 中加一个 1 以表示接下来要往右
68             node_sign_list = node_sign_list[:depth] + [1]

```

接下来要做的，就是实现关键的信息整合的部分了。具体而言，当我们在先序遍历的过程中遇到了叶节点的话，就需要做如下几件事情：

- 统计“有效的”中间节点。
- 按照“权值符号和相应输出值的符号一致，并把权值大小设为决策路径上神经元个数的倒数”的规则，将第1隐藏层、第2隐藏层之间的权值设置好。
- 根据叶节点中的类别分布，将第2隐藏层与输出层之间的权值设置好。

其中第一步中所谓的“有效的”中间节点，是针对决策树在实际生成过程中的一些冗余所提出来的。我们在前 3 节进行讨论时，暂时把“所有决策面都会垂直于坐标轴”的限制给去掉了，所以才能得到如图 4.4 所示的“完美的”决策树；但是如果把这个限制加上的话，生成出来的决策树可能就是相对较“丑”的决策树了。事实上以 `scikit-learn` 为例，它在图 4.3 所示的 X 数据集上的表现将如图 4.11 所示。

可以发现，实际生成出来的决策树的决策面都是“横平竖直”的，完全不是理想中的简单的两条直线。而且如果把视角拉远的话，就会看到更加糟糕的结果，如图 4.12 所示。

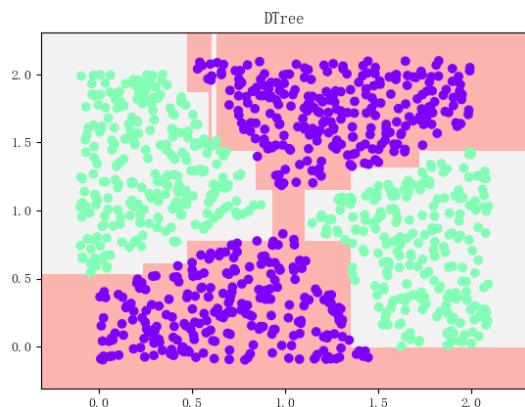


图4.11 `scikit-learn`决策树在X数据集上的表现（1）

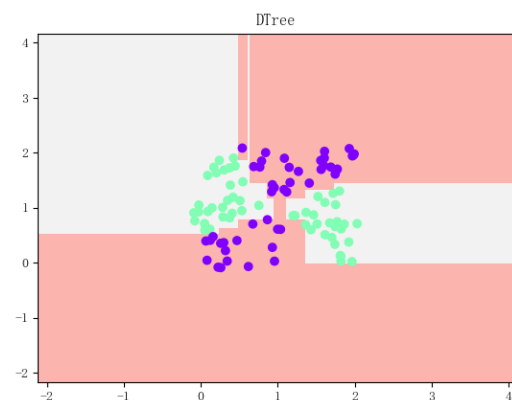


图4.12 `scikit-learn`决策树在X数据集上的表现（2）

导致该结果的原因仍然是 `scikit-learn` 决策树的决策面是垂直于坐标轴的。所以在这种情况下，为了用垂直于坐标轴的决策面去拟合“真实的”、不垂直于坐标轴的分类超平面（或分类曲面）时，决策树就不得不做很多额外的事情。比如，此时决策树中的一条决策路径就完全有可能是：“ $x^{(1)} < -1 \rightarrow x^{(2)} \geq 0 \rightarrow x^{(1)} < -1.5 \rightarrow y = 0$ ”。那么此时就能看到，决策路径中的第一项——“ $x^{(1)} < -1$ ”——所对应的中间节点，对于该决策路径而言就是一个“冗余的”中间节点。这是因为该决策路径最后的中间节点为 $x^{(1)} < -1.5$ ，而该条件已经包含了 $x^{(1)} < -1$ 的逻辑。

因此我们在整合信息时，为了让模型更加精简，需要把这些冗余的中间节点去除，而这就是所谓的“统计有效的中间节点”。

那么在知道了信息整合的整套逻辑后，我们就能进行具体的实现了。

```

69         # 如果先序遍历到了叶节点的话，就开始整合信息
70     else:
71         # 将有效的中间节点用一个集合 valid nodes 来存储
72         valid_nodes = set()
73         # 将 node sign list 做一个局部复制以方便去除冗余节点
74         local_sign_list = node_sign_list[:]
75         # 遍历储存着决策路径上节点信息的两个列表
76         for i, (
77             (node_id, node_dim, node_threshold),
78             node_sign

```

```

79         ) in enumerate(zip(node list, node sign list)):
80             # 将节点信息加入 valid nodes
81             valid nodes.add((node id, node sign))
82             if i >= 1:
83                 # 从第2个节点开始, 检查之前的节点是否是冗余的节点
84                 for j, (
85                     (local id, local dim, local threshold),
86                     local sign
87                 ) in enumerate(zip(
88                     node list[:i], local sign list[:i]
89                 )):
90                     # 如果发现当前节点和之前某个节点的
91                     # 符号与特征维数都一致, 那么就进行
92                     # 进一步的检查
93                     if node sign == local sign and node dim == local dim:
94                         # 如果都是 $x^{(i)} < \bar{\epsilon}_i$  (或 $\geq \bar{\epsilon}_i$ ) 的形式
95                         # 且当前节点比之前节点的阈值 $\bar{\epsilon}_i$ 
96                         # 更小 (或更大) 的话, 就意味着
97                         # 之前的节点是冗余的节点
98                         if ((
99                             node sign == -1 and
100                             node threshold < local threshold
101                         ) or (
102                             node sign == 1 and
103                             node threshold > local threshold
104                         )):
105                             # 对于冗余的节点, 进行相应的删减操作
106                             local sign list[j] = 0
107                             valid nodes.remove((local id, local sign))
108                             break
109             # 将第1隐藏层与第2隐藏层之间的权值设置好
110             for node id, node sign in valid nodes:
111                 w2[node id, leaf id cursor] = node sign / len(valid nodes)
112             # 更新最大决策路径长度
113             max_route_length = max(max_route_length, len(valid_nodes))
114             # 将第2隐藏层与输出层之间的权值设置好
115             w3[leaf_id_cursor] = rs / np.sum(rs)
116             # 最后, 把 leaf_id_cursor 加1, 表示我们又
117             # “见过了”一个叶节点
118             leaf_id_cursor += 1

```

在完成了最困难的信息整合步骤后, 就只需要再做一些收尾的工作了。

```

119         # 给第1隐藏层到第2隐藏层的权值乘上最大决策路径长度
120         # 这么做的原因会在 4.3.1 节中给出, 这里暂时按下不表
121         w2 *= max route length
122         # 将各个权值矩阵储存在 self. transform ws 中
123         self. transform ws = [w1, w2, w3]
124         # 由于我们只用设置第1隐藏层中的偏置量
125         # 所以 self. _transform_bs 中只需储存一个偏置量 b

```

```
126 self._transform_bs = [b]
```

以上我们就完成了整套决策树转神经网络的代码实现，其具体效果如图 4.13 所示。

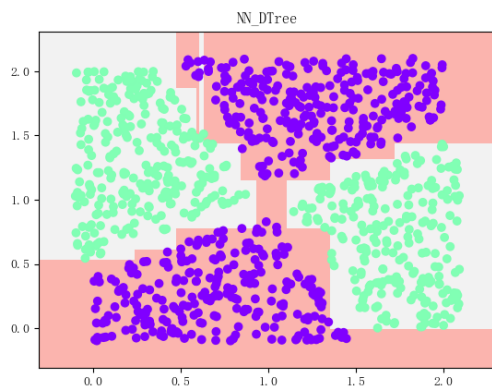


图4.13 NN_{DT} 在X数据集上的表现

可以看到，这张图和图 4.11 所示的 `scikit-learn` 的决策树的表现是完全一致的，这在一定程度上证明了我们转换算法的正确性。

4.3 模型转换的实际意义

在前两节中，我们具体介绍了如何将朴素贝叶斯与决策树模型转换成神经网络，不过为什么要这样做的原因则还没有进行太多的说明。虽然我们的初衷是想借此说明神经网络的强大，但将传统算法转换为神经网络的意义显然并不应局限于此。所以在这一节中，我们会对模型转换的实际意义进行简要的说明。

注意：为了叙述方便，我们进行如下两个符号的约定。

- NN_{NB} ：代指由朴素贝叶斯转换而得的神经网络模型。
- NN_{DT} ：代指由决策树转换而得的神经网络模型。

具体而言，在前两节讨论模型的转换算法时，为了保证转换的等价性，我们对激活函数做了比较强的限制。比如对于 NN_{NB} 来说，我们在输出层没有使用变换函数来对模型的原始输出做一个变换，对于 NN_{DT} 而言则限制了第 1 隐藏层、第 2 隐藏层的激活函数为我们平时基本不会用到的两个激活函数：`sign` 函数与 `one_hot` 函数。但是如果只从神经网络自身的角度来看问题的话，完全可以使用各式各样的激活函数并期望神经网络能够表现得更好。所以在 4.3.1 和 4.3.2 节中，我们会分别从朴素贝叶斯的角度和决策树的角度，来说明如何通过改变激活函数来赋予模型全新的意义以及更好的性能。

另一方面，由前文的讨论可知，无论是朴素贝叶斯还是决策树，转换算法的本质其实就是对神经网络的权值矩阵、偏置量以及激活函数进行特定的设置。那么换个角度来看的话，我们完全可以将这种特定的设置当作参数初始化的方法，然后接下来完全按照神经网络的思维走。这就意味着在将传统机器学习算法转换为神经网络后，可以进一步使用神经网络中的反向传播

算法来微调所得到的新模型。理想情况下，这种微调应该能够在一定程度上缓解原始模型的不良特性，从而给转换后的神经网络带来更好的性质。因此在 4.3.3 和 4.3.4 节中，我们同样会分别从朴素贝叶斯的角度和决策树的角度，来说明这种微调为什么是行之有效的。

注意：核心的测试代码可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/d_Traditional2NN/NaiveBayes2NN.ipynb（对应 NN_{NB} ）和 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/d_Traditional2NN/DTree2NN.ipynb（对应 NN_{DT} ）。

4.3.1 利用 Softmax 来赋予概率意义

在这节中，我们首先来看看如何改进 NN_{NB} 。由第 2 章的讨论可知，朴素贝叶斯是做了条件独立性假设的，这个过强的假设导致其模型的最终表现与模型的初衷不尽相符。仍以 K 分类问题为例，朴素贝叶斯的最初想法是构造一个概率模型 P ，它能真实地反映标签 y 在特征向量 \mathbf{x} 条件下的后验概率：

$$P(\mathbf{x}) = (p(y = c_1|\mathbf{x}), \dots, p(y = c_K|\mathbf{x}))^T$$

从而模型的最终输出为

$$G(\mathbf{x}) = \arg \max_k P(\mathbf{x})_k = \arg \max_k p(y = c_k|\mathbf{x})$$

但是在做了条件独立性假设后，朴素贝叶斯构造出来的概率模型 P 就变成了

$$P(\mathbf{x})_k = \frac{1}{p(\mathbf{x})} \cdot p(y = c_k) \prod_{i=1}^n p(x^{(i)}|y = c_k)$$

而由于 $\frac{1}{p(\mathbf{x})}$ 不影响最终输出时 $\arg \max$ 的取值，所以朴素贝叶斯最终得到的概率模型其实就变成了：

$$P(\mathbf{x})_k = p(y = c_k) \prod_{i=1}^n p(x^{(i)}|y = c_k)$$

这里面的一个巨大的鸿沟是，朴素贝叶斯最初想设计的概率模型 P 确实是一个概率模型，但最终得到的模型已经不是概率模型了。具体而言，它最初想设计的 P 是满足下面这个性质的：

$$\sum_{k=1}^K P(\mathbf{x})_k = 1$$

但是最终得到的 P 却不满足该性质。换句话说，朴素贝叶斯的初衷是设计出一个原始输出就是概率输出的模型，不过在做了条件独立性假设的“妥协”以后，朴素贝叶斯最终的原始输出就不再是概率输出了。

对于这个问题，解决的方法其实比较直观——直接在输出层用变换函数 **Softmax** 来将它变成概率输出即可。不过，虽然这个方法看上去简单粗暴，但它确实是有相应理论作为支撑的，

下面我们就来看看其背后真正的逻辑。

首先来复习一下 Softmax 函数的形式：

$$\phi^{(m)}(\mathbf{h}) = (\varphi^{(1)}(h^{(1)}), \varphi^{(2)}(h^{(2)}), \dots, \varphi^{(K)}(h^{(K)}))^T$$

其中

$$\varphi^{(i)}(h^{(i)}) = \frac{e^{h^{(i)}}}{\sum_{j=1}^K e^{h^{(j)}}}$$

换句话说，它是先将原始输出的 $h^{(i)}$ 变成了 $e^{h^{(i)}}$ ，然后直接对它们进行归一化处理。表面上来看的话，使用 e 的指数形式只是为了让各个值都不小于 0，从而满足概率的基本性质，但是其背后真正的逻辑其实是：它将原始输出的 $h^{(i)}$ 视为真实概率的对数值。换句话说，假设真实的概率输出是

$$\mathbf{p} = (p^{(1)}, p^{(2)}, \dots, p^{(K)})^T$$

那么 Softmax 函数就认为，模型的原始输出 \mathbf{h} 其实是 \mathbf{p} 的对数值：

$$\mathbf{h} = \log \mathbf{p} \Rightarrow h^{(i)} = \log p^{(i)}$$

从而经过 Softmax 的作用后，就能把 \mathbf{h} 转化为 \mathbf{p} （注意 $\sum_{i=1}^K p^{(i)} = 1$ ）：

$$\varphi^{(i)}(h^{(i)}) = \frac{e^{h^{(i)}}}{\sum_{j=1}^K e^{h^{(j)}}} = \frac{e^{\log p^{(i)}}}{\sum_{j=1}^K e^{\log p^{(j)}}} = \frac{p^{(i)}}{\sum_{i=1}^K p^{(i)}} = p^{(i)}$$

那么在 Softmax 的这种思想下，结合朴素贝叶斯的线性形式，其实就能发现很有意思的事情。由 4.1 节的讨论可知，为了得到朴素贝叶斯的线性形式，我们要使用对数函数来转化模型的表达式。那么对应到朴素贝叶斯的最初想法的话，就是要将原概率模型的公式转化为

$$P(\mathbf{x}) = \log \mathbf{p}(y|\mathbf{x} = c_k)$$

换句话说，在实际的操作中，我们本来就是将模型的原始输出视为真实概率的对数值的！这就使得应用 Softmax 作为变换函数成为不仅自然而且严谨的做法。

4.3.2 利用 Tanh+Softmax 来“软化”模型

在这一节中，我们会讨论如何对 NN_{DT} 做出改进。正如前文所说，为了保证转换的等价性，我们接连使用了 sign 函数与 one_hot 函数这两个“非主流”的激活函数。而事实上，由于决策树的决策面都是垂直于坐标轴的，所以为了让神经网络也表达出这一点，这两个激活函数本身都是非常“硬”的。下面就来直观地解释一下什么叫作“硬”，对于理论上的解释则会放在下一节中进行。

注意：所谓的“硬”，本身其实是没有严格定义的，更多情况下它只是一种直观的感受。所以即使我们会在后文通过公式的推导来解释何为“硬”，也只是为了让大家能更好地从直观上去把握这个感觉而已。

为了进行直观的解释，我们需要用到具体的例子。第3章用过的九宫格数据集就是一个很好的选择，因为决策树以及转换而得的神经网络在图3.14所示的九宫格数据集上的表现都将如图4.14所示。

可以看到，模型几乎把最为正确的规律给学出来了，但是这个“最为正确的规律”本身却不一定是合理的。具体而言，九宫格数据集中有一些非常模糊的分界点——比如 $(-1, -1)$ 、 $(-1, 1)$ 、 $(1, 1)$ 和 $(1, -1)$ ——它们的四周同时遍布了三种不同类别的样本，如图4.15所示，每块区域中的数字代表着属于该区域的样本的标签，从而中间的四个交叉点就是四个非常模糊的分界点。

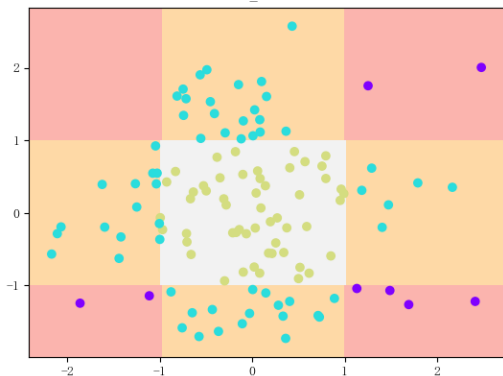


图4.14 决策树与相应的 NN_{DT} 在九宫格数据集上的表现

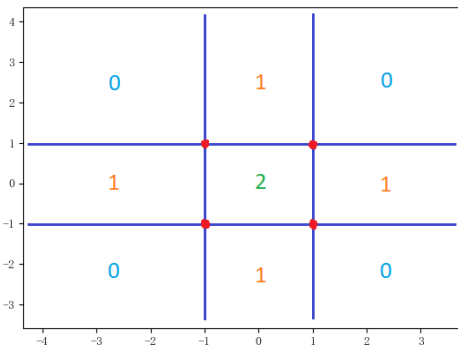


图4.15 九宫格数据集中模糊的分界点

对于这些分界点而言，图4.14所示的模型非常严格地将四周的样本进行了正确分类。然而对于实际的问题而言，如果问题本身并没有那么“硬派”的分类条件，只不过是一些噪声的影响导致了它看上去很“硬”，那么如图4.14所示的模型毫无疑问就陷入了过拟合的窘境。而这种分界面过“硬”问题的主要根源就在于我们选取的激活函数太硬了，毕竟对于 sign 函数而言，它的函数图像将如图4.16所示。

可以看到它在0点处有一个很大的“跳跃”，在非零点处则又是恒定不变的值，从而就有一种“非黑即白”的感觉。而对于 one_hot 这个多元函数来说，它只会把输入的 n 个数中最大的那个数保留下来并作为相应的输出，对于其余 $n-1$ 个输入所对应的输出则统一设为0，从直观上来说就是“一枝独秀”。这两个激活函数各自本身就已经足够硬了，如今我们还把它们放在一起使用，毫无疑问就会得到一个相当硬的神经网络模型。

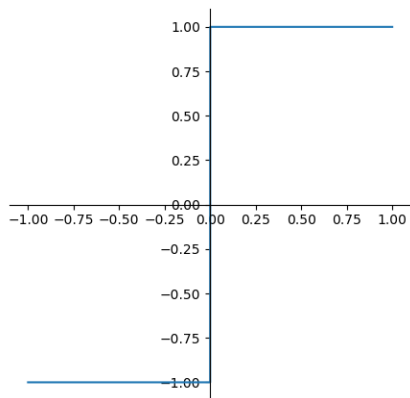


图4.16 sign 函数的函数图像

那么为了解决这种“过硬”的问题，自然的想法就是让它“软化”，用数学的语言来说就是构造它们的“平滑函数”，使得它们从不可导的函数变成可导函数。具体而言，对于 `sign` 函数来说，我们可以用第 3 章介绍过的 `Tanh` 函数来替换它。`sign` 函数和 `Tanh` 函数的比较如图 4.17 所示。

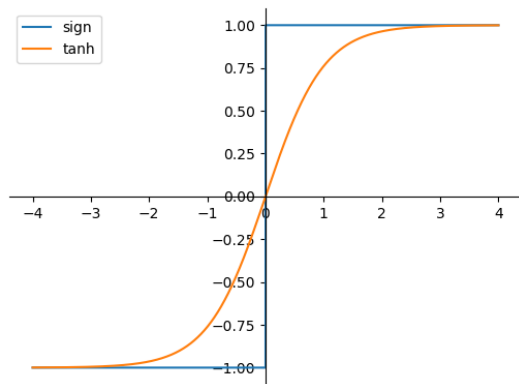


图4.17 `sign`与`Tanh`的比较

可以看到，与 `sign` 函数直接从-1跳跃到 1 不同，`Tanh` 在-1到 1 之间采用了平滑的曲线来过渡，从而达到了“软化”的效果。

同样的，对于 `one_hot` 函数而言，也有一个我们很熟悉的、能够软化它的函数——`Softmax`。在 4.2.3 节我们曾经说过，第 2 隐藏层的激活函数需要满足“除了对各个输入中的最大值进行保留以外、其余输入值都进行较大幅度的衰减”这样一个性质。而对一个 n 维的特征向量 \mathbf{x} 而言，由于 `Softmax` 函数的输出为

$$\varphi^{(i)}(\mathbf{x}^{(i)}) = \frac{e^{x^{(i)}}}{\sum_{j=1}^n e^{x^{(j)}}}$$

所以当对于 \mathbf{x} 两个维度的特征 $x^{(i)}$ 与 $x^{(j)}$ 而言，它们之间的差距在经过 `Softmax` 的作用后，是会以指数级的速度放大的，换句话说就是，非最大值相比起最大值而言会以指数级的速度衰减。由于指数级速度的衰减完全可以算是“较大幅度的衰减”，所以 `Softmax` 是满足第 2 隐藏层对激活函数的要求的。

此外，为了保证不发生数值溢出，我们在计算 `Softmax` 函数之前，一般会先将 \mathbf{x} 的 n 个维度都减去 \mathbf{x} 这 n 个维度中的最大值。具体而言，假设

$$x^{(m)} = \max[x^{(1)}, x^{(2)}, \dots, x^{(n)}]$$

那么就将 \mathbf{x} 转换为 $\tilde{\mathbf{x}}$

$$\tilde{\mathbf{x}} \triangleq (x^{(1)} - x^{(m)}, x^{(2)} - x^{(m)}, \dots, x^{(n)} - x^{(m)})^T$$

此时不难发现，`Softmax` 的输出将满足

$$\varphi^{(i)}(x^{(i)}) = \frac{e^{x^{(i)}}}{\sum_{j=1}^n e^{x^{(j)}}} = \frac{e^{-x^{(m)}} \cdot e^{x^{(i)}}}{e^{-x^{(m)}} \cdot \sum_{j=1}^n e^{x^{(j)}}} = \frac{e^{x^{(i)} - x^{(m)}}}{\sum_{j=1}^n e^{x^{(j)} - x^{(m)}}} = \varphi^{(i)}(\tilde{x}^{(i)})$$

而且会有

$$\varphi^{(m)}(\tilde{x}^{(m)}) = \varphi^{(m)}(0) = \frac{1}{\sum_{j=1}^n e^{\tilde{x}^{(j)}}}$$

所以如果 $\tilde{x}^{(j)}$ 都比较小的话，那么 $\varphi^{(m)}(\tilde{x}^{(m)})$ 的输出将会趋近于 1，从而使得 Softmax 函数甚至满足“除了指定决策路径 l_i 对应的神经元的输出 $o(v_i)$ 接近 1 以外，其余神经元的输出接近 0”这样一个比较强的性质。而这也正是为什么我们在代码 4.4 的 109 行处对第 1 隐藏层和第 2 隐藏层之间的权值矩阵 \mathbf{w}_2 乘上了一个最大决策路径长度，因为这样就能够放大最大值与其余值之间的差距，从而使得上述较强的性质能被更好地满足。

那么在将 sign+one_hot 的组合替换成 Tanh+Softmax 的组合后，神经网络在九宫格数据集上的决策边界将如图 4.18 所示。

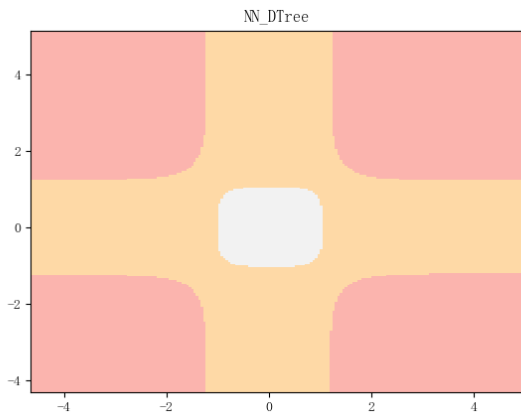


图 4.18 Tanh+Softmax 的 NN_{DT} 在九宫格数据集上的决策边界

可以看到，在权值矩阵与偏置量都没变的情况下，通过改变激活函数，我们就将原模型的“硬”决策边界软化了。从直观上来说，在各个模糊的分界点处，新的神经网络都采取了类似投票的机制来进行最终的分类；附近哪个类别的样本更多，模型就更倾向于输出那个对应的类别。

注意：这种性质并不一定总是好的。虽然我们前面说图 4.14 所示的模型有可能陷入过拟合，但即使是在现实任务中，我们也不能排除各个模糊的分界点恰好都是最为关键的“支持点”，或者训练集分布与测试集分布确实一致的可能性。只不过从概率的角度来看，像图 4.15 所示的模糊的分界点一般而言确实应该是模糊的（“软”的）而不是确凿的（“硬”的）。

为了更直观地说明软化的效果（以及转换算法本身的正确性），我们可以把 NN_{DT} 在九宫格数据集上的、各个神经元的具体激活情况画出来，如图 4.19 和图 4.20 所示。可以看到，第 1 隐藏层中各个神经元的激活情况确实表达出了决策树中的超平面，而第 2 隐藏层中各个神经元的激活情况也确实表达出了决策树中各个决策路径所划分出来的特征子空间。同时，由于无论

选取怎样的激活函数， NN_{DT} 所需要表达的超平面都是一致的，可以看到在图 4.19、图 4.20 中，第 1 隐藏层的激活情况是一样的；而由于使用 Tanh 可以软化决策边界，这意味着此时各个决策路径划分出来的特征子空间的边界将会比使用 sign 时的边界更平滑，因此可以看到图 4.20 中第 2 隐藏层的激活情况会比图 4.19 中的更平滑。

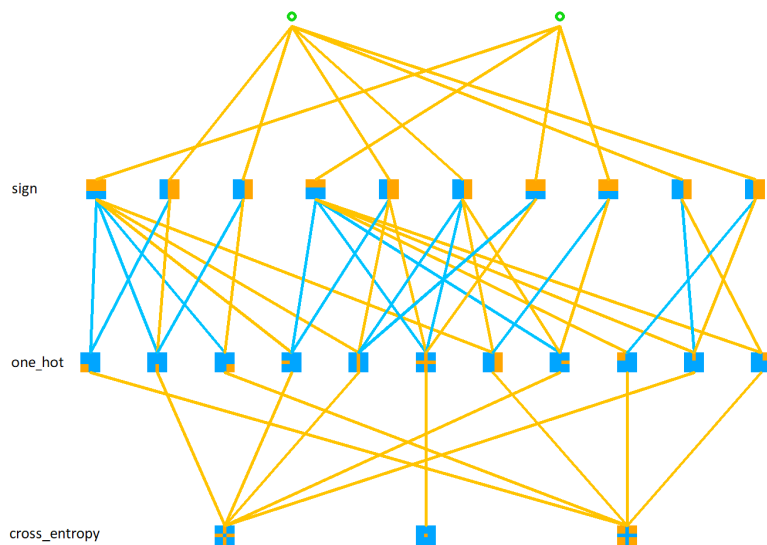


图4.19 sign+one_hot的 NN_{DT} 在九宫格数据集上的激活情况

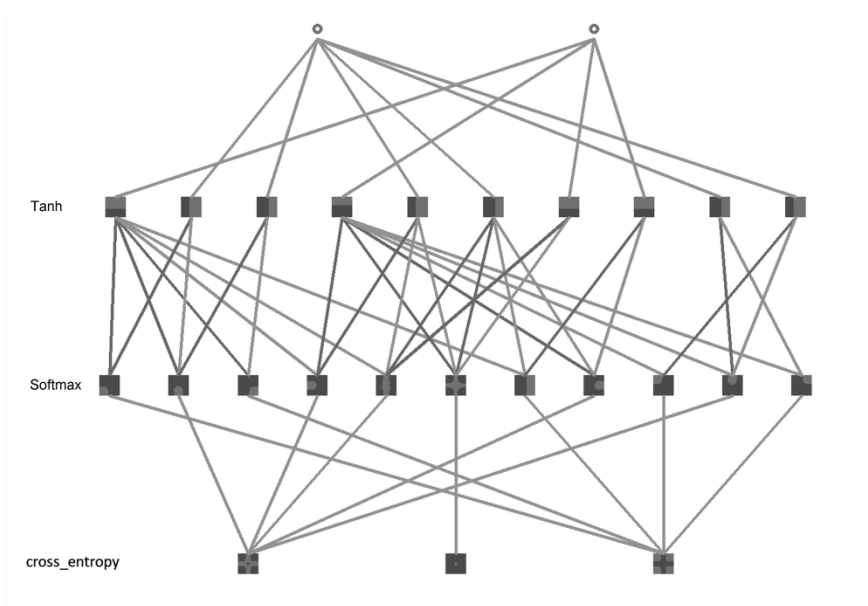


图4.20 Tanh+Softmax的 NN_{DT} 在九宫格数据集上的激活情况

以上我们讨论了如何通过增加或改变激活函数来赋予转换后的模型以全新的意义，接下来会探讨为何在此基础上，利用神经网络本身的训练方法——反向传播算法来微调模型，可以让转换后的模型更有效。

4.3.3 通过微调来缓解“条件独立性假设”

下面就先来看看微调 NN_{NB} 的作用。在朴素贝叶斯中，我们最想通过微调来缓解的东西，毫无疑问就是它的条件独立性假设了。注意，在上一节中我们曾经说过，利用 Softmax 函数能够让神经网络更贴近于朴素贝叶斯的想法，所以我们就基于使用了 Softmax 作为变换函数的神经网络来展开讨论。由 4.1 节可知，神经网络的原始输出为

$$h(\mathbf{x}) = \mathbf{W}_0 + \mathbf{W}\mathbf{x} = \begin{bmatrix} \log p(y = c_1) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_1) \\ \log p(y = c_2) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_2) \\ \vdots \\ \log p(y = c_K) + \sum_{i=1}^n x^{(i)} \log p(x^{(i)}|y = c_K) \end{bmatrix}$$

从而在经过了 Softmax 之后，神经网络的概率输出即为

$$\begin{aligned} o(\mathbf{x})_k &= \varphi^{(k)}(h(\mathbf{x})_k) = \frac{e^{h(\mathbf{x})_k}}{\sum_{j=1}^n e^{h(\mathbf{x})_j}} \\ &= \frac{p(y = c_k) \prod_{i=1}^n p(x^{(i)}|y = c_k)^{x^{(i)}}}{\sum_{j=1}^n p(y = c_j) \prod_{i=1}^n p(x^{(i)}|y = c_j)^{x^{(i)}}} \end{aligned}$$

换句话说，条件独立性假设其实蕴含在了用 Softmax 作为变换函数的神经网络的概率输出中。那么如果我们用反向传播算法对模型进行训练的话，由于原始模型的对数先验概率（偏置量）和对数条件概率（权值矩阵）都会被梯度下降法改变，所以概率输出中相应的“概率”—— $p(y = c_k)$ 与 $p(x^{(i)}|y = c_k)$ ——的意义就不再是原始的先验概率与条件概率了，而是某些由神经网络学习出来的“隐概率”了。换句话说，此时我们的神经网络没有拘泥于条件独立性假设，而是学出了新的概率表达形式。

4.3.4 通过微调来丰富超平面的选择

然后来看看对 NN_{DT} 进行微调的意义。我们在前文已经反复强调过，决策树算法有“超平面垂直于坐标轴”的缺点。反映在数学公式上的话，就是决策树中所有中间节点所对应的超平面的表达式都会形如 $x^{(i)} - \tilde{\epsilon}_i = 0$ 。由 NN_{DT} 的构造算法的第一步可以看出，该不良特性在神经网络中的具体表现，在于我们会将输入层到第 1 隐藏层的权值矩阵设置为一个非常稀疏的矩阵。具体而言，假设我们的特征向量是三维的特征向量

$$\mathbf{x} = (x^{(1)}, x^{(2)}, x^{(3)})^T$$

然后决策树中一共有三个超平面

$$a_1 x^{(1)} - \tilde{\epsilon}_1 = 0, \quad a_2 x^{(3)} - \tilde{\epsilon}_2 = 0, \quad a_3 x^{(2)} - \tilde{\epsilon}_3 = 0$$

假设它们从左到右依次对应着 NN_{DT} 第 1 隐藏层中的第 1 个到第 3 个神经元，那么输入层到第 1 隐藏层的权值矩阵就是

$$\mathbf{W}^{(1)} = \begin{bmatrix} a_1 & 0 & 0 \\ 0 & 0 & a_2 \\ 0 & a_3 & 0 \end{bmatrix}$$

类似的，如果有 n 个特征而决策树中一共有 m 个超平面的话，那么 $\mathbf{W}^{(1)}$ 就将是一个 $m \times n$ 的矩阵，且该矩阵的每一行都有且仅有一个元素非 0。这种稀疏性从数学上就大大地限制了第 1 隐藏层所能表达的超平面的能力（具体而言就是使得第 1 隐藏层只能表达垂直于坐标轴的超平面）。

那么如果通过反向传播算法来进一步训练的话，是否能消除这种稀疏性呢？这个问题的讨论就能从理论上解释为何 `sign` 函数和 `one_hot` 函数都是“硬”函数了。事实上，由第 3 章的反向传播算法的推导可以得出：

$$\delta^{(1)} = \mathbf{W}^{(2)\top} \delta^{(2)} \odot \phi'(h^{(1)})$$

其中，由于我们用了 `sign` 函数作为第 1 隐藏层的激活函数，所以上式中的 $\phi^{(1)}$ 其实就是 `sign` 函数。此外由图 4.16 不难看出，`sign` 函数在 0 点不可导（因为它“跳跃”了一下），在其余点的导数都为 0（因为它在其余点取的都是恒定值），从而就有

$$\phi'(h^{(1)}) \equiv 0, \quad \forall h^{(1)}$$

于是

$$\delta^{(1)} = \mathbf{W}^{(2)\top} \delta^{(2)} \odot \phi'(h^{(1)}) \equiv 0$$

另一方面，偏置量 $\mathbf{W}_0^{(1)}$ 与权值矩阵 $\mathbf{W}^{(1)}$ 的梯度表达式分别为

$$\nabla_{\mathbf{W}_0^{(1)}} L = \delta^{(1)}, \quad \nabla_{\mathbf{W}^{(1)}} L = \delta^{(1)} \times \mathbf{o}^{(0)\top}$$

从而 $\mathbf{W}_0^{(1)}$ 和 $\mathbf{W}^{(1)}$ 的梯度就总会取 0，这就意味着无论再怎么进行训练， $\mathbf{W}_0^{(1)}$ 和 $\mathbf{W}^{(1)}$ 都不会被更新。换句话说，如果使用 `sign` 函数作为第 1 隐藏层的激活函数的话，那么模型所能表达的超平面不仅仍然只能是垂直于坐标轴的超平面，而且还会和初始化得到的超平面完全一致。这事实上就失去了“微调”的意义，从而也正是为何我们之前说 `sign` 函数很硬的原因——只要我们对第 1 隐藏层用了 `sign` 函数作为激活函数，那么第 1 隐藏层的表达能力就定死了，再没有调整的可能。

再来看为何 `one_hot` 函数是硬的函数。由于 `one_hot` 是用在第 2 隐藏层的激活函数，所以我们用 $\phi^{(2)}$ 来代指它。由前文可知，假设 $\phi^{(2)}$ 的输入为

$$\mathbf{h}^{(2)} = (h^{(2)(1)}, h^{(2)(2)}, \dots, h^{(2)(n^{(2)})})^\top$$

且 $h^{(2)(i)}$ 是这 $n^{(2)}$ 个数中最大的数，即

$$h^{(2)(i)} = \max_j h^{(2)(j)}$$

的话，那么 $\phi^{(2)}$ 的表达式即为

$$\phi^{(2)}(\mathbf{h}^{(2)}) = (0, \dots, h^{(2)(i)}, \dots, 0)^T$$

从而

$$\phi'^{(2)}(\mathbf{h}^{(2)}) = (0, \dots, \overset{i}{1}, \dots, 0)^T$$

此时可知

$$\delta^{(2)} = \mathbf{W}^{(3)T} \delta^{(3)} \odot \phi'^{(2)}(\mathbf{h}^{(2)}) = (0, \dots, \overset{i}{\delta^{(2)}}, \dots, 0)^T$$

其中, $\delta^{(2)}$ 代指 $\mathbf{W}^{(3)T} \delta^{(3)}$ 的第 i 个数字。那么由

$$\nabla_{\mathbf{W}_0^{(2)}} L = \delta^{(2)}, \quad \nabla_{\mathbf{W}^{(2)}} L = \delta^{(2)} \times \mathbf{o}^{(1)T}$$

不难看出, $\nabla_{\mathbf{W}_0^{(2)}} L$ 是一个 OneHot 梯度, 而 $\nabla_{\mathbf{W}^{(2)}} L$ 则是除了第 i 行为 $\delta^{(2)} \cdot \mathbf{o}^{(1)T}$ 、其余行都为 0 的梯度。这意味着在模型训练的每次迭代中, $\mathbf{W}_0^{(2)}$ 、 $\mathbf{W}^{(2)}$ 分别只有第 i 个元素和第 i 行被更新, 其余元素、行都会被按住不动。

one_hot 函数的这种性质本身其实就已经相当硬了, 而如果此时第 1 隐藏层用的激活函数还是 sign 函数的话, 会导致更硬的结果。具体而言, 由于

$$\mathbf{h}^{(2)} = \mathbf{W}_0^{(2)} + \mathbf{W}^{(2)} \mathbf{o}^{(1)}$$

以及

$$\mathbf{o}^{(1)} = \phi^{(1)}(\mathbf{h}^{(1)}) = \phi^{(1)}(\mathbf{W}_0^{(1)} + \mathbf{W}^{(1)} \mathbf{o}^{(0)})$$

那么由于 sign 函数会导致 $\mathbf{W}_0^{(1)}$ 和 $\mathbf{W}^{(1)}$ 都无法被更新, 因此 $\mathbf{o}^{(1)T}$ 相对于输入 \mathbf{x} 而言就可以视为一个固定值。结合前文的讨论, 就能发现权值矩阵 $\nabla_{\mathbf{W}^{(2)}} L$ 每次不仅只有第 i 行非 0, 而且这个非 0 行的方向也是固定的 (因为在 $\delta^{(2)} \cdot \mathbf{o}^{(1)T}$ 中, $\delta^{(2)}$ 是一个标量, 只有 $\mathbf{o}^{(1)}$ 是一个向量, 所以决定方向的其实只有 $\mathbf{o}^{(1)}$)。换句话说, 此时神经网络第 1 隐藏层与第 2 隐藏层之间的权值矩阵 $\mathbf{W}^{(2)}$ 每次的更新, 其实都只是它的第 i 行朝着某个固定的方向 ($\mathbf{o}^{(1)T}$) 来做更新, 不同之处仅仅表现在 i 的不同以及更新速率 ($\delta^{(2)}$) 的不同而已, 这无疑就使得模型更加“硬”了。

所以, 选用 Tanh+Softmax 不仅能像 4.3.2 节直观叙述的那样将决策边界软化, 它还能从理论上软化神经网络的训练算法——反向传播算法。事实上在使用了 Tanh+Softmax 之后, 由于各个参数都能正常地完成更新, 所以输入层与第 1 隐藏层之间的权值矩阵将不再会局限于稀疏矩阵, 而可以是任意矩阵, 从而意味着此时神经网络就能够运用第 1 隐藏层来表达出任意的超平面了。

事实上在使用了 Tanh+Softmax 后, NN_{DT} 在图 4.3 所示的 X 数据集上的初始表现将如图 4.21 所示。

可以看到它和图 4.13 所示的结果类似, 不同点主要在于图 4.21 中的决策边界是比较光滑的, 而图 4.13 中的决策面是比较硬的。同时仅从初始表现来看, 图 4.21 其实是比不过图 4.13 的。

但正如本节所讨论的, 使用 Tanh+Softmax 除了能够如 4.3.2 节所说的那样“磨平”决策边

界以外，它还允许我们利用反向传播算法来进行模型的微调。而当我们微调完模型后，它的表现将如图 4.22 所示。

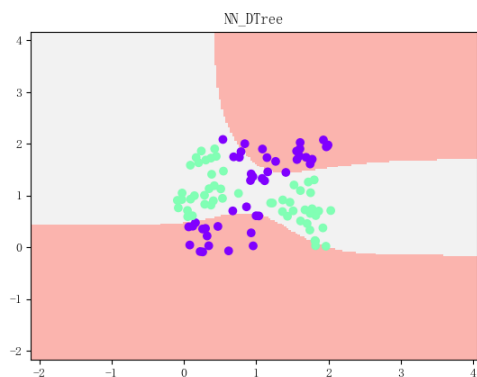


图4.21 Tanh+Softmax的 NN_{DT} 在X数据集上的初始表现

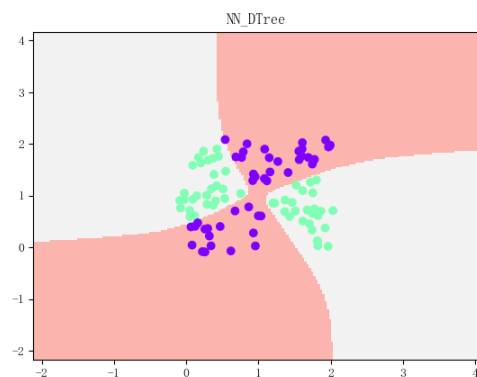


图4.22 Tanh+Softmax的 NN_{DT} 在X数据集上微调后的表现

可以看到，虽然这个结果也不尽善尽美，但是已经比图 4.13 所示的原始结果要好上不少了。此外我们曾经说过，数据量对神经网络而言是非常重要的，所以我们可以尝试把数据量加大并期望神经网络能表现得更好。事实上，图 4.22 所示的结果是 1000 量级样本的结果。如果我们把样本量级提升到 10 万的话，决策树与使用了 Tanh+Softmax 并进行微调后的 NN_{DT} 的表现将分别如图 4.23 和图 4.24 所示。

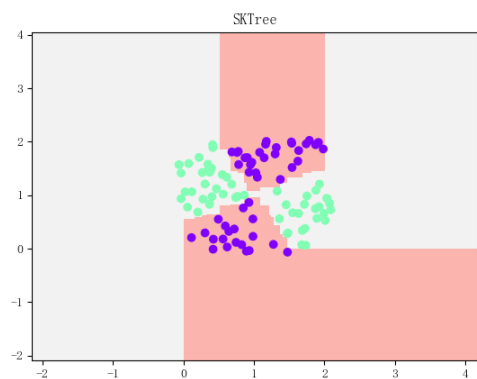


图4.23 决策树在10万量级的X数据集上的表现

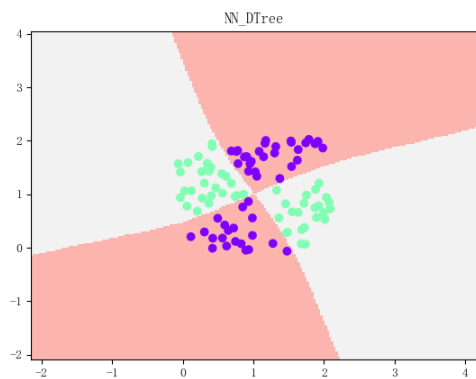


图4.24 Tanh+Softmax的 NN_{DT} 在10万量级的X数据集上微调后的表现

可以看到，决策树仍然陷入了很强的过拟合，但 NN_{DT} 则已经几乎把“真实的规律”给拟合出来了。

最后想要指出的是，sign 函数和 one_hot 函数并非一无是处，在特定的场合也拥有特定的功能。比如在 2.2.3 节的最后就曾经提到过，利用神经网络可以对决策树进行类似于后剪枝的操作，下面我们就来介绍如何利用 sign+Softmax 的组合来做到这一点。

由前文的讨论可知，sign 函数能够保持住决策树原有的超平面不变，且 NN_{DT} 中第 1 隐藏层和第 2 隐藏层之间的权值矩阵 $W^{(2)}$ 能反映出决策路径。而决策树后剪枝的本质，其实是将某个中间节点转化为叶节点，并将后续分化出的决策路径提前终止。这表现在权值矩阵中的话，其

实就是简单地调整某些权值的大小而已。具体而言,假设在某条决策路径 l_i 上有 n 个中间节点,这 n 个中间节点对应着 NN_{DT} 第1隐藏层中 $u_1 \sim u_n$ 这 n 个神经元。那么如果我们用第2隐藏层中的 v_i 神经元来表示决策路径 l_i 的话,由前文的讨论即可知, $u_1 \sim u_n$ 到 v_i 的权值将分别为

$$u_1 \sim u_n \rightarrow v_i : \left[o(u_1) \cdot \frac{1}{n}, o(u_2) \cdot \frac{1}{n}, \dots, o(u_n) \cdot \frac{1}{n} \right]$$

那么假设我们在第 m 个中间节点处进行了局部剪枝的话,就只需把上述权值更改为

$$u_1 \sim u_n \rightarrow v_i : \left[o(u_1) \cdot \frac{1}{m}, o(u_2) \cdot \frac{1}{m}, \dots, o(u_m) \cdot \frac{1}{m}, 0, \dots, 0 \right]$$

即可。注意到对第 m 个中间节点进行局部剪枝将不仅影响到 l_i 这一条决策路径,所以需要对所有影响到的决策路径对应的权值都做一个调整。

总之我们证明了,调整 $\mathbf{W}^{(2)}$ 中的权值大小确实能表达出后剪枝这个操作。那么如果我们微调 sign+Softmax 对应的 NN_{DT} 的话,就会发现:

- 由于在第1隐藏层用了 sign 函数,所以所有超平面都不会被改变。
- 由于在第2隐藏层用了 Softmax 函数,所以我们可以对 $\mathbf{W}^{(2)}$ 中的所有权值大小做出更新(虽然更新的方向仍然是固定的 $\mathbf{o}^{(1)T}$)。

换句话说,sign+Softmax 对应的 NN_{DT} 从理论上确实是蕴含了后剪枝的功能的,我们确实能够期望通过微调该模型来近似完成对原决策树的后剪枝操作。为了更有说服力,下面就用 UCI 上一个比较著名的人造数据集 Madelon 来佐证这一点。

Madelon 数据集的详细介绍可以在 <http://archive.ics.uci.edu/ml/datasets/madelon> 这个链接中获得,完整的数据集则可以参见 https://github.com/carefree0910/MachineLearning/blob/master/_Data/madelon.txt,这里我就大概总结与翻译一下主要内容。Madelon 是 5 个 NIPS 2003 特征提取挑战数据集中的,它有 5 个原始特征和这些特征的 15 个线性组合(即总共有 20 个有效特征),此外还有 480 个无用特征(或说噪声)。样本量方面,Madelon 的训练集有 2000 个样本,交叉验证集则有 600 个。由于提交测试结果的网站似乎已经无法访问了,所以我们就直接用交叉验证集上的结果作为衡量模型好坏的标准。

首先来看看决策树的表现与 NN_{DT} 的初期表现,如图 4.25 所示。为方便叙述,我们用 acc_{tr} 来代指训练准确率、用 acc_{te} 来代指测试准确率。

```
=====
Decision Tree performance
-----
acc - Train :      1.0  CV : 0.763333
-----
Internals : 152 ; Leafs : 153
=====
Initial performance
-----
acc - Train :      1.0  CV :      0.765  Test : None
-----
Epoch  0  Iter      0  Snapshot      0 (acc) - Train :      1.0  Test :      0.765
Epoch  1  Iter     15  Snapshot      3 (acc) - Train :      1.0  Test : 0.768333
Epoch  3  Iter     45  Snapshot      9 (acc) - Train :      0.98  Test :      0.77
Epoch  5  Iter     70  Snapshot     14 (acc) - Train :      0.995  Test :      0.77
Epoch  5  Iter     75  Snapshot     15 (acc) - Train :      1.0  Test : 0.771667
Epoch 10  Iter    150  Snapshot     30 (acc) - Train :      0.995  Test :      0.775
```

图4.25 决策树与sign+Softmax的 NN_{DT} 在Madelon数据集上的表现

可以看到，决策树的表现是 $\text{acc}_{\text{tr}} = 100\%$ 、 $\text{acc}_{\text{te}} = 76.3\%$ ，而 sign+Softmax 的 NN_{DT} 的初始化表现则是 $\text{acc}_{\text{tr}} = 100\%$ 、 $\text{acc}_{\text{te}} = 76.5\%$ 。换句话说，无论是决策树还是初始 NN_{DT} 的表现，它们都存在非常严重的过拟合现象。

注意：之所以决策树的表现与初始 NN_{DT} 的表现不完全一致，是因为我们在第 2 隐藏层中用了 Softmax 函数来代替等价转换中的 one_hot 函数。

因此如果微调 NN_{DT} 确实能够起到后剪枝的作用的话，那么在训练过程中就应该能观察到测试准确率的提升。从图 4.26 中我们能看到，至少在初期阶段的训练中， NN_{DT} 的测试准确率的确是在不断提升的。而事实上当 NN_{DT} 完成了微调之后，它的表现将如图 4.26 所示。

Epoch	52	Iter	780	Snapshot	156 (acc)	- Train :	1.0	Test :	0.795
Epoch	53	Iter	795	Snapshot	159 (acc)	- Train :	1.0	Test :	0.796667
Epoch	57	Iter	855	Snapshot	171 (acc)	- Train :	1.0	Test :	0.796667
Epoch	58	Iter	870	Snapshot	174 (acc)	- Train :	1.0	Test :	0.798333
Epoch	62	Iter	930	Snapshot	186 (acc)	- Train :	1.0	Test :	0.796667
Epoch	65	Iter	975	Snapshot	195 (acc)	- Train :	1.0	Test :	0.8
Epoch	67	Iter	1005	Snapshot	201 (acc)	- Train :	1.0	Test :	0.8
Epoch	70	Iter	1050	Snapshot	210 (acc)	- Train :	1.0	Test :	0.801667
Epoch	72	Iter	1080	Snapshot	216 (acc)	- Train :	1.0	Test :	0.801667
Epoch	75	Iter	1125	Snapshot	225 (acc)	- Train :	1.0	Test :	0.805
Epoch	77	Iter	1155	Snapshot	231 (acc)	- Train :	1.0	Test :	0.803333
Epoch	82	Iter	1230	Snapshot	246 (acc)	- Train :	1.0	Test :	0.805
Epoch	86	Iter	1290	Snapshot	258 (acc)	- Train :	1.0	Test :	0.805

图4.26 sign+Softmax的 NN_{DT} 在Madelon数据集上微调后的表现

也就是说， NN_{DT} 的最终表现能达到 $\text{acc}_{\text{tr}} = 100\%$ 、 $\text{acc}_{\text{te}} = 80.5\%$ ，这是一个可喜的提升。

为了让实验结果更有说服力，我们再用相同结构的 BasicNN 做一下实验。由图 4.25 可以看出，决策树的中间节点、叶节点分别有 152、153 个，即我们用来做对比实验的神经网络的第 1 隐藏层、第 2 隐藏层的神经元个数也应该分别是 152 和 153 个。在这种设置下，当激活函数都使用 ReLU（即默认激活函数）时，神经网络的表现将如图 4.27 所示。

Epoch	33	Iter	485	Snapshot	97 (acc)	- Train :	0.49	Test :	0.55
Epoch	33	Iter	490	Snapshot	98 (acc)	- Train :	0.525	Test :	0.558333
Epoch	33	Iter	495	Snapshot	99 (acc)	- Train :	0.555	Test :	0.555
Epoch	34	Iter	500	Snapshot	100 (acc)	- Train :	0.535	Test :	0.558333
Epoch	34	Iter	505	Snapshot	101 (acc)	- Train :	0.49	Test :	0.543333
Epoch	34	Iter	510	Snapshot	102 (acc)	- Train :	0.505	Test :	0.538333
Epoch	35	Iter	515	Snapshot	103 (acc)	- Train :	0.51	Test :	0.546667
Epoch	35	Iter	520	Snapshot	104 (acc)	- Train :	0.51	Test :	0.545
Epoch	35	Iter	525	Snapshot	105 (acc)	- Train :	0.575	Test :	0.531667
Epoch	36	Iter	530	Snapshot	106 (acc)	- Train :	0.51	Test :	0.545
Epoch	36	Iter	535	Snapshot	107 (acc)	- Train :	0.535	Test :	0.5
Epoch	36	Iter	540	Snapshot	108 (acc)	- Train :	0.53	Test :	0.5
Epoch	37	Iter	545	Snapshot	109 (acc)	- Train :	0.5	Test :	0.5

图4.27 BasicNN在Madelon数据集上的表现

可以看到，朴素的神经网络连训练集都无法很好地拟合（ $\text{acc}_{\text{tr}} \approx 55\%$ ），测试集的表现就更不用提了（ $\text{acc}_{\text{te}} \approx 53\%$ ）。这一方面是因为特征冗余量太大，另一方面则是因为样本量太小。这两个问题对于神经网络本身而言是致命的，但是对决策树而言却不是太大的问题。因为决策树在每次挑选特征作为划分依据时都会看它带来多少信息量，而由于冗余的特征理论上不会带来信息量，所以决策树就不会挑选冗余的特征作为划分依据。同时我们还曾经提过，无论样本量有多少，理论上决策树都能在训练集上达到 100% 的准确率，因为它只需要不断细分特征向量空间即可。因此对于 Madelon 数据集来说，使用决策树的关键问题在于如何进行适当的剪枝以

防止过拟合。那么从图 4.26 所示的结果来看的话，微调 NN_{DT} 确实能够起到一定的剪枝效果。

注意：以上两点原因都是 BasicNN 无法在测试集上有良好表现的原因，而至于 BasicNN 在训练集上表现如此之差的根本原因则在于没有对原始数据进行标准化处理，我们会在第 5 章的第 1 节里叙述相应的细节。

此外，笔者把在 Madelon 数据集上进行实验的相关测试专用代码放在了 GitHub 上，感兴趣的读者可以参阅 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/Madelon/TestDT2NN.py。

4.3.5 模型逆转换的可能性

在前文的所有讨论中，我们都把注意力放在了如何将传统算法转换成神经网络上。如果使用“逆转思维”来看待这个转换过程的话，就会发现存在着进行模型逆转换的可能性。

先来看看 NN_{NB} 的逆转换过程。对于一般的退化后的神经网络（即线性模型）

$$o(\mathbf{x}) = \phi(\mathbf{W}_0 + \mathbf{W}\mathbf{x})$$

来说，假设在经过梯度下降法的训练后，模型参数收敛到了 $\widehat{\mathbf{W}}_0$ 和 $\widehat{\mathbf{W}}$ ，其中

$$\begin{aligned}\widehat{\mathbf{W}}_0 &= (\widehat{W}_0^{(1)}, \widehat{W}_0^{(2)}, \dots, \widehat{W}_0^{(K)})^T \\ \widehat{\mathbf{W}} &= \begin{bmatrix} \widehat{W}_{11} & \widehat{W}_{12} & \dots & \widehat{W}_{1n} \\ \widehat{W}_{21} & \widehat{W}_{22} & \dots & \widehat{W}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \widehat{W}_{K1} & \widehat{W}_{K2} & \dots & \widehat{W}_{Kn} \end{bmatrix}\end{aligned}$$

此时如果我们令

$$p_k \triangleq \log \frac{e^{\widehat{W}_0^{(k)}}}{\sum_{k=1}^K e^{\widehat{W}_0^{(k)}}}$$

以及

$$p_{ki} \triangleq \log \frac{e^{\widehat{W}_{ki}}}{\sum_{k=1}^K e^{\widehat{W}_{ki}}}$$

并且按照 NN_{NB} 的思想，把 p_k 视为各个类别的对数先验概率，并把 p_{ki} 视为各个类别下相应特征的对数条件概率的话，就有

$$\log p(y \in c_k) = p_k$$

以及

$$p(x^{(i)} | y \in c_k) = p_{ki}$$

从而就有

$$p(y \in c_k) = \frac{e^{\widehat{W}_0^{(k)}}}{\sum_{k=1}^K e^{\widehat{W}_0^{(k)}}}$$

以及

$$p(x^{(i)}|y \in c_k) = \frac{e^{\hat{W}_{ki}}}{\sum_{k=1}^K e^{\hat{W}_{ki}}}$$

换句话说，通过把退化后的神经网络（线性模型）逆转换为朴素贝叶斯，我们就可能可以获得一定程度上的模型的概率意义。

而对于 NN_{DT} 而言，逆转换是类似的。我们完全可以先训练一个普通的三层神经网络，然后把第 1 隐藏层的神经元视为诸多超平面、把第 2 隐藏层的神经元视为诸多决策路径。只不过由于 NN_{DT} 的等价转换算法有许多限制条件（比如激活函数的限制），所以相应的逆转换会比 NN_{NB} 对应的逆转换要困难很多。但是，一旦能够近似地完成逆转换的话，那么神经网络就能享受决策树那极强的可解释性了。

4.4 模型转换的局限性

我们在上一节叙述了不少将传统算法转换成神经网络的意义，但除了 4.3.5 节所说的意义以外，其余意义大多是基于传统算法而言的。换句话说，虽然进行模型转换对于提升传统算法的性能而言可能确实有不少帮助，但是对于神经网络本身而言的意义却可能没那么大。具体而言，无论我们怎样修改 NN_{NB} 模型，它的本质都只是线性模型而已，即它的表达能力总是会受到很大的限制；而对于 NN_{DT} 模型来说，虽然在上一节我们展示了许多看上去挺优雅的性质，但该模型的结构设计还是有些太过“刻意”了，显得不够自然。

此外，在真实场景所对应的任务中，好的算法一般很少会只训练单个决策树模型，而是会训练多个决策树模型并做一个集成，比如著名的 Random Forest (https://en.wikipedia.org/wiki/Random_forest) 和 xgboost (<https://xgboost.readthedocs.io/en/latest/>)。此时如果把所有训练出来的决策树都转换成神经网络的话，一方面模型复杂度会骤然提升很多倍，另一方面则是这些好的算法所蕴含的优良性质很有可能会被神经网络的微调给破坏掉，而不进行微调的话转换的意义又很有限，从而意味着进行模型转换的性价比可能会比较低。

总之，如果我们能够设计出一种神经网络的结构，使得它不仅能天生蕴含决策树的表达能力，而且还能兼顾一定的剪枝与集成的能力的话，那么无疑会比这种直接转换而得的模型要更为出色。令人振奋的是，这样的神经网络结构确实是存在的。而至于如何具体地把它实现出来，就是接下来第 5 章所要介绍的内容了。

4.5 本章小结

- 朴素贝叶斯的本质是线性模型，从而它能被神经网络表达。
- 决策树的本质是用（垂直于坐标轴的）超平面划分特征向量空间，从而它能被神经网络表达。具体而言：

- 神经网络的第1隐藏层对应着各个超平面。
- 神经网络的第2隐藏层对应着各个决策路径。
- 神经网络的输出层对应着各个叶节点。
- 通过在输出层使用 **Softmax**, 能使 NN_{NB} 拥有概率意义。如果在此基础上微调模型的话, 就能在一定程度上缓解条件独立性假设。
- 将第1隐藏层、第2隐藏层的激活函数换成 **Tanh+Softmax** 能“软化”模型。如果在此基础上微调模型的话, 就能让神经网络的第1隐藏层表达出任意的超平面(而不一定是垂直于坐标轴的)。
- 微调 **sign+Softmax** 的 NN_{DT} 能在一定程度上达到后剪枝的效果。
- 直接从原始传统算法转换而得的神经网络是有不少局限性的。

第 5 章

神经网络进阶

在第 4 章中，通过设计神经网络的权值矩阵、偏置量与激活函数，我们成功地把第 2 章中介绍的朴素贝叶斯与决策树都转换为了神经网络。不过，这个转换的本质其实只是在进行特殊的参数初始化，它并没有为神经网络引入新的东西。这导致的一个问题就是，虽然从形式上来看， NN_{NB} 与 NN_{DT} 都是神经网络，但它们实质上与一般的神经网络还是有着比较大的区别的。对于 NN_{NB} 而言，由于它是线性模型，这直接导致它无法被称为深度学习的工具（因为与“深度”二字相违背）；对于 NN_{DT} 而言，则是人工的痕迹太过严重，那过于稀疏的初始化权值矩阵是基本不可能被正常地训练出来的。换句话说，如果用常规的训练方式，我们是无法得到 NN_{DT} 的结构，这就使得 NN_{DT} 和主流的神经网络之间有种割裂感。

不过，这些局限性的存在也是必然而且合理的：我们毕竟只是在朴素的神经网络的参数初始化上做了修改，朴素神经网络（BasicNN）的诸多根本性的问题其实并没有在这个过程中得到解决，比如模型表达能力太强以至于容易陷入过拟合、反向传播中梯度过于敏感、网络模型的结构过于单一，以及层与层之间全连接的连接形式等。如果想让神经网络在实际问题中有好的表现的话，那么这些根本性的问题是必须解决的。所以本章的主要目的就是介绍针对这些问题的解决方案。具体而言，我们将会在第 5.1 节介绍 Dropout 与 Batch Normalization 这两个经常会使用到的技巧，在第 5.2 节、5.3 节介绍 Wide and Deep 与 DNDF 这两个结构，在第 5.4 节介绍如何对神经网络进行剪枝，并在第 5.5 节介绍这些技术之间的互补性。而在最后的第 5.6 节，我们则会做许多实验来验证这些技术的有效性。

需要指出的是，本章所涉及的所有技术都属于“增强版神经网络（AdvancedNN）”的一部分，所以其实所有的代码都是 AdvancedNN 的代码实现中的一部分。如果读者想直接看到完整的实现的话，可以参见 https://github.com/carefree0910/MachineLearning/blob/master/_Dist/NeuralNetworks/e_AdvancedNN/NN.py，笔者也强烈建议大家在看本章的代码时对照着完整的实现来看，因为这样就能对整体的实现思路有一个更好的把握。

值得一提的是，本章所介绍的技术不仅能应用在（全连接）神经网络上，也能应用在卷积神经网络（CNN）与循环神经网络（RNN）上。由于这些技术在 CNN、RNN 上应用的方式和本章所将介绍的、在（全连接）神经网络上应用的方式别无二致，所以我们不会具体介绍 CNN、RNN 相关的应用方式，而是将关注的重点放在（全连接）神经网络上。

本章主要涉及的知识点有：

- 神经网络的训练技巧
- 神经网络的结构设计
- 神经网络的剪枝技术

5.1 层结构内部的额外工作

对于朴素的神经网络而言，层结构内部只需要用激活函数作用于输入并得到输出即可。但是如果考虑到神经网络的特性的话，就有必要在层结构内部进行一些额外的工作以使得神经网络能在实际任务中发挥作用。

5.1.1 Dropout

在这一节中，我们先来看看 BasicNN 很容易过拟合的问题。由 Universal Approximation Theorem 可知，神经网络的表达能力是毋庸置疑的。虽然该理论没有保证可学习性，但是它毕竟提供了一个扎实的根基。因此我们常常需要担心的并不是神经网络学不出东西，而是神经网络学的东西太过片面。这也是为何在许多情况下，神经网络在训练集上的准确率能达到 100%，而在测试集上的准确率却只有 60%~70%。

注意：在第 4 章用到过的原始 Madelon 数据集上，BasicNN 的训练准确率其实也很低。其中的原因会在本节后面介绍 Batch Normalization 时详细解释，这里暂且按下不表。

由于第 1 章和第 2 章介绍决策树剪枝时都曾说过，防止过拟合的非常普遍的做法是对模型 G 的复杂度做出一定的惩罚，从而使 G 趋于精简。在这种思想下，Srivastava 等人在 *Journal of Machine Learning Research* 15 (2014) 上的一篇论文中首先提出了一个很有意思的技巧——Dropout（原论文共 30 页，感兴趣的读者可以参见 <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>）。简单来说的话，Dropout 会在每次迭代中依概率（通常在命名时会以 `keep_prob` 来命名）保留对应层结构中的某些神经元，并将没有保留的神经元去掉，从而使得每次迭代中实际训练的其实都是小的、精简的“子神经网络”，如图 5.1 所示。

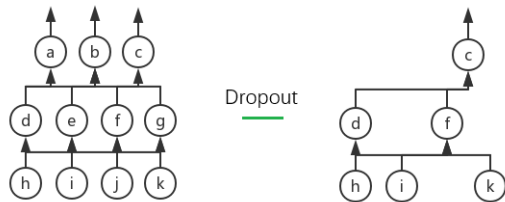


图 5.1 所示的即为当 `keep_prob` 设为默认

图 5.1 Dropout 示意图

值 50% 时, 某次迭代中 Dropout 的一种可能的表现。图 5.1 左图所示为原网络, 右图所示为 Dropout 后的子网络, 可以看到, 只有神经元 c、d、f、h、i、k 被保留了下来, 神经元 a、b、e、g、j 都被 Drop 掉了。

Dropout 过程的合理性需要概率论上的一些理论作为支撑, 不过鉴于 TensorFlow 框架有封装好的相应函数, 我们就只从直观上去说明一下它的运作机理 (以 keep_prob 为 50% 为例, 其余 keep_prob 的情况是同理的):

- 在训练过程中, 由于 Dropout 后留下来的神经元可以理解为 “在 50% 的概率下幸存下来” 的神经元, 所以将它们对应的输出进行 “增幅” 是合理的。具体而言, 假设一个神经元 n 的输出本来是 o , 那么如果 Dropout 后它被留下来的话, 其输出就应该变成 $o \times \frac{1}{50\%} = 2o$ 。换句话说, 我们应该让带 Dropout 的期望输出和原输出一致: 对于任意一个神经元 n , 设 keep_prob 为 p 而其原输出为 o , 那么当带 Dropout 的输出为 $o \times \frac{1}{p}$ 时, n 的期望输出即为 $p \times o \times \frac{1}{p} = o$ 。
- 由于在训练时已经保证了神经网络的期望输出不变, 所以在预测过程中还是应该让整个网络一起进行预测而不进行 Dropout。这不仅能让模型的统计意义更好, 还能让模型的表现更为稳定 (因为 Dropout 的随机性太强)。

实验证明, 在使用了 Dropout 之后, 神经网络的过拟合现象确实得到了很大的缓解。此外, 通过调整 Dropout 的 keep_prob, 我们甚至可以调整防止神经网络过拟合的 “力度”。具体而言, 如果神经网络过拟合得很严重的话, 就可以通过调小 keep_prob 来让每次迭代中训练的子神经网络更精简; 如果神经网络过拟合现象比较轻微的话, 就可以通过将 keep_prob 调大来让每次留存下来的、能被训练的神经元变多, 从而加快训练的速度。

值得一提的是, 对于我们曾在第 3 章介绍过的 SELU 函数而言, 如果既想使用 Dropout, 又想要让它的 “能使每个神经元的输出值都趋向于服从一个均值为 0、方差为 1 的标准正态分布” 这个性质保留的话, 我们就需要对相应的 Dropout 做出适当的变动。这种专门为 SELU 设计的 Dropout 叫作 “alpha dropout”, 在 SELU 的原论文 (<https://arxiv.org/pdf/1706.02515.pdf>) 中给出了具体的公式, TensorFlow 中也有相应的实现 (https://github.com/tensorflow/tensorflow/blob/Book/tensorflow/contrib/nn/python/ops/alpha_dropout.py), 感兴趣的读者可以查阅相应的部分。

5.1.2 Batch Normalization

在本节里, 我们再来看看神经网络对输入数据有特殊要求的问题。由于神经网络的表达能力大多来自激活函数的非线性性, 所以让输入数据 “适应于” 激活函数是很重要的。具体而言:

- 对于 Sigmoid 系列的激活函数而言, 由于函数两端过于平坦, 所以为了不陷入梯度消失的窘境, 我们希望神经元的输入集中在函数中央, 而不希望神经元的输入 (的绝对值) 过大。
- 对于 ReLU 激活函数而言, 由于函数在输入小于 0 的区域恒为 0, 那么从直观上来说, 如果一个很大的梯度把某个神经元的输入拉到了小于 0 很多的区域, 该神经元从此以

后的输出就将永远是 0 了，因为它基本不可能回到大于 0 的区域。这就是著名的“Dying ReLU 问题”，如果不针对它做足准备的话，通常会导致最后有很多神经元都处于“死亡”的状态——它们的输出恒为 0，而且无论再怎么训练，也无法将它们的输出拉回正值。

- 对于其他常用的激活函数而言，通常也有输入越小于 0，梯度越接近于 0 的性质。这些函数虽然不至于像 ReLU 那么极端，但或多或少都可能会因为一个巨大的梯度流将输入推到小于 0 很多的区域，从而导致出现神经元死亡的问题。

总结一下，可以把激活函数对输入数据的要求归结为如下两点：

- 输入数据不要过大。
- 输入数据带来的梯度也不要太大。

注意到由第 3 章的局部梯度公式

$$\delta^{(i)} = \mathbf{w}^{(i+1)\top} \delta^{(i+1)} \odot \phi'^{(i)}(\mathbf{h}^{(i)})$$

以及大多数（非 Sigmoid 系列的）激活函数 $\phi^{(i)}$ 在输入数据 $\mathbf{h}^{(i)}$ 比较大时都有

$$\phi'^{(i)}(\mathbf{h}^{(i)}) \approx 1$$

可知， $\delta^{(i)}$ 随着反向传播算法是以权值矩阵 $\mathbf{w}^{(i)}$ 的指数级速度在增长的，这就很容易出现我们不想见到的、巨大的梯度流。所以总体上来说，我们希望数据处在合理的区间内，不要有太大的异常值。因此一个合理且行之有效的做法就是：

- 将输入数据的均值控制在 0，从而意味着数据的“中心”是 0。
- 将输入数据的方差控制在 1，从而意味着数据的“波动”不会太大。

这就是之前曾反复提到过的数据标准化。数据标准化的背后有许多很好的数学性质，相关的推导也有些复杂，不过它的思想却可以这样直观地去理解：对于神经网络乃至任何机器学习分类器来说，其目的都可以说是从训练样本集中学出样本在样本空间中的分布，从而可以利用这个分布来预测未知数据所属的类别。而具体到神经网络模型的话，由于我们常常采用的是 MBGD 算法，所以模型在每次迭代中看到的数据，其实仅仅是训练样本集中的一个 Mini Batch。如果不对每个 Mini Batch 进行任何操作的话，由于训练集本身只是样本空间中的一个小抽样，而 Mini Batch 又只是训练集的一个小抽样，所以不难想象它们彼此对应的“极大似然分布（极大似然估计意义下的分布）”是各不相同的。这样的话，分类器在接受每个 Mini Batch 时都要学习一个新的分布，然后在最后还要尝试从这些不同的分布中总结出样本空间的总分布，这无疑是相当困难的。因此如果存在一种处理方法能够使每个 Mini Batch 的分布都贴近真实分布的话，对分类器的训练来说无疑是至关重要的。而标准化，其实正是经典的一种处理方法。

为了能更直观地感受标准化的重要性，我们拿 Madelon 数据集来做一个实验。在 4.3.4 节的最后，我们曾用过 Madelon 的原始数据来训练 BasicNN，结果发现模型甚至无法拟合好训练集（ $\text{acc}_{\text{tr}} \approx 55\%$ ），测试集的表现自然也就很差（ $\text{acc}_{\text{te}} \approx 53\%$ ）。如果我们亲眼去看 Madelon 数据集的特征向量的话，就可以看到它每个维度的特征的取值都非常大（基本都是几百的数值），

这显然不是神经网络所“喜欢”的输入。而如果我们先对 Madelon 数据集做一个标准化，即令神经网络的数据预处理函数 $\phi^{(0)}$ 为

$$\phi_0(\mathbf{x}) = \frac{\mathbf{x} - \text{mean}}{\text{std}}$$

的话，就会发现神经网络的表现将如图 5.2 所示。

Epoch	41	Iter	615	Snapshot	41 (acc)	-	Train :	1.0	Test : 0.563333
Epoch	42	Iter	630	Snapshot	42 (acc)	-	Train :	1.0	Test : 0.561667
Epoch	43	Iter	645	Snapshot	43 (acc)	-	Train :	1.0	Test : 0.561667
Epoch	44	Iter	660	Snapshot	44 (acc)	-	Train :	1.0	Test : 0.561667
Epoch	45	Iter	675	Snapshot	45 (acc)	-	Train :	1.0	Test : 0.561667
Epoch	46	Iter	690	Snapshot	46 (acc)	-	Train :	1.0	Test : 0.563333
Epoch	47	Iter	705	Snapshot	47 (acc)	-	Train :	1.0	Test : 0.563333
Epoch	48	Iter	720	Snapshot	48 (acc)	-	Train :	1.0	Test : 0.565
Epoch	49	Iter	735	Snapshot	49 (acc)	-	Train :	1.0	Test : 0.563333
Epoch	50	Iter	750	Snapshot	50 (acc)	-	Train :	1.0	Test : 0.565
Epoch	51	Iter	765	Snapshot	51 (acc)	-	Train :	1.0	Test : 0.563333
Epoch	52	Iter	780	Snapshot	52 (acc)	-	Train :	1.0	Test : 0.565
Epoch	53	Iter	795	Snapshot	53 (acc)	-	Train :	1.0	Test : 0.565

图5.2 BasicNN在标准化后的Madelon数据集上的表现

可以看到，此时 BasicNN 就已经能在训练集上做到 100%的准确率了。虽说在测试集上比之前没有好太多 ($\text{acc}_{\text{te}} \approx 56.3\%$)，但这至少证明了神经网络的拟合能力是可以保证的。

注意：测试代码可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/Madelon/TestAdvancedNN.py。

总之，神经网络比较喜欢“中规中矩”的输入，所以标准化是有必要的。但是需要指出的是，仅仅对原始输入数据进行标准化是不充分的。因为虽然这种做法能保证输入数据的质量，但它却无法保证神经网络隐藏层的输入数据的质量。试想神经网络中的第 1 隐藏层 L_1 ，它接收的输入 $\mathbf{h}^{(1)}$ 是输入层 L_0 的输出 $\mathbf{o}^{(0)} = \phi^{(0)}(\mathbf{x})$ 和权值矩阵 $\mathbf{W}^{(1)}$ 相乘后再加上偏置量 $\mathbf{W}_0^{(1)}$ 后的结果。在训练过程中，虽然由于我们将数据预处理函数 $\phi^{(0)}$ 设置为了标准化函数，从而能够保证 $\mathbf{o}^{(0)}$ 的质量，但由于 $\mathbf{W}^{(1)}$ 和 $\mathbf{W}_0^{(1)}$ 在训练过程中会不断地被更新，所以 $\mathbf{h}^{(1)}$ 的分布其实仍然不断地在发生变化。换句话说， $\mathbf{h}^{(1)}$ 的质量其实已经没有了保证了。

于是，为了解决这种随着前向传导算法的推进，得到的数据质量会不断变差的问题，Sergey Ioffe 和 Christian Szegedy 在 2015 年最先提出了 Batch Normalization（常简称为 BN）的技巧，它通过对中间层数据进行某种“规范化”处理，从而达到类似对输入进行标准化处理的效果。需要指出的是，简单地将每层得到的数据进行直接的标准化操作显然是不可行的，因为这样会破坏每层自身学到的数据特征。设想如果某一层 L_i 学到了“数据基本都分布在样本空间的边缘”这一特征，那么如果强行做标准化处理并把数据弄成均值为 0、方差为 1 的“中心化”数据的话，无疑就摒弃了 L_i 所学到的、可能是非常有价值的知识。

为了使“规范化”之后不破坏层结构本身学到的特征，BN 采取了一个简单却十分有效的方法：引入两个可以学习的“重构参数”以期能够从规范化的数据中重构出层本身学到的特征。具体参见算法 5.1。

算法 5.1 BN 的规范化过程

输入：某一层 L_i 在当前 Mini Batch 上的输出 $\mathbf{o}^{(i)}$ 、增强数值稳定性所用的小值 ϵ

过程：

(1) 计算当前 Mini Batch 数据的均值、方差：

$$\mu_i = \overline{\mathbf{o}^{(i)}}$$

$$\sigma_i^2 = \text{var}(\mathbf{o}^{(i)})$$

(2) 标准化：

$$\widehat{\mathbf{o}^{(i)}} = \frac{\mathbf{o}^{(i)} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

(3) 线性变换：

$$\widetilde{\mathbf{o}^{(i)}} = \gamma \widehat{\mathbf{o}^{(i)}} + \beta$$

输出：规范化处理后的输出 $\widetilde{\mathbf{o}^{(i)}}$

BN 的核心即在于 γ 、 β 这两个参数的应用上。关于如何利用反向传播算法来更新这两个参数的数学推导不算很难但有点多，这里就不展开叙述了，感兴趣的读者可以参见原论文 <https://arxiv.org/abs/1502.03167> 来了解相应的细节。对于本节而言，我们会直接利用已经封装好的 TensorFlow 函数来进行实现。

需要指出的是，对于算法中均值和方差的计算其实还有一个被广泛使用的小技巧，该小技巧在某种意义上可以说是用到了“动量”的思想：我们会分别维护两个储存“滚动均值（Running Mean）”和“滚动方差（Running Variance）”的变量。具体参见算法 5.2。

算法 5.2 BN 的规范化过程（Momentum Ver.）

输入：

1) 某一层 L_i 在当前 Mini Batch 上的输出 $\mathbf{o}^{(i)}$

2) 增强数值稳定性所用的小值 ϵ

3) 动量值 m （一般取 $m = 0.9$ ）

过程：

首先要初始化 Running Mean、Running Variance 为 0 向量：

$$\mu_{\text{run}} = \sigma_{\text{run}}^2 = 0$$

并初始化 γ 、 β 为 1、0 向量：

$$\gamma = 1, \quad \beta = 0$$

然后进行如下操作：

(1) 计算当前 Mini Batch 数据的均值、方差：

$$\mu_i = \overline{\mathbf{o}^{(i)}}$$

$$\sigma_i^2 = \text{var}(\mathbf{o}^{(i)})$$

(2) 利用 μ_i 、 σ_i^2 和动量值 m 更新 μ_{run} 、 σ_{run}^2 ：

$$\mu_{\text{run}} \leftarrow m \cdot \mu_{\text{run}} + (1 - m) \cdot \mu_i$$

$$\sigma_{\text{run}}^2 \leftarrow m \cdot \sigma_{\text{run}}^2 + (1 - m) \cdot \sigma_i^2$$

(3) 利用 μ_{run} 、 σ_{run}^2 来标准化输出：

$$\widehat{\mathbf{o}^{(i)}} = \frac{\mathbf{o}^{(i)} - \mu_{\text{run}}}{\sqrt{\sigma_{\text{run}}^2 + \epsilon}}$$

(4) 线性变换：

$$\widetilde{o^{(i)}} = \gamma o^{(i)} + \beta$$

输出：规范化处理后的输出 $\widetilde{o^{(i)}}$

最后提两点使用 BN 时需要注意的事项：

- 无论是算法 5.1 还是算法 5.2，BN 的训练过程和预测过程的表现都是不同的。具体而言，训练过程和算法中所叙述的一致，此时的均值和方差都是根据当前的 Mini Batch 来计算的；但在测试过程中，我们的均值和方差就不能根据当前 Batch 来计算，而应该根据训练样本集的某些特征来进行计算。对于算法 5.2 来说， μ_{run} 和 σ_{run}^2 天然就是很好的、可以用来当测试过程中的均值和方差的变量，对于算法 5.1 而言需要额外的计算。
- 对于使用了 BN 的层结构来说，线性映射中的偏置量会变成一个冗余的变量。这是因为规范化操作中的标准化这一步会让输入数据减去它的均值，这就会直接将偏置量的影响抹去。同时由于 BN 本身的 β 参数可以说正是破坏对称性的参数，所以它能比较好地完成原本偏置量所做的工作。
- BN 并不是万金油，它那将每层的输出都进行标准化的做法并不一定和其他依赖于每层原始输出来调优神经网络的技术相兼容（比如本章后面将介绍的 DNDF 与软剪枝技术）。

5.1.3 具体实现

前两节我们在理论层面上对 Dropout 和 Batch Normalization 做了介绍，下面就来看看如何具体地实现它们。由于 TensorFlow 有相应的封装，所以这两者的实现其实都几乎只是一行代码的事。注意，这两个技巧都是在层结构内部进行的，所以它们都应该实现在 `self.build_layer` 这个方法中。

```
01 def build_layer(self, i, net):
02     # 如果设置为使用 BN 的话，就直接利用 TensorFlow 的相应函数
03     # 对层结构的输入进行规范化处理。注意我们这里用到了
04     # 代码 3.1 中提过的 self._is_training 这个标识着
05     # 当前是否是训练过程的属性
06     if self.use_batch_norm:
07         net = tf.layers.batch_normalization(
08             net, training=self._is_training, name="BN{}".format(i))
09         activation = self.activations[i]
10         # 处理激活函数
11         if activation is not None:
12             net = getattr(Activations, activation)(
13                 net, "{}{}".format(activation, i))
14         # 如果设置为使用 Dropout（即 keep_prob 设置为小于 1 的值）
15         # 就直接利用 TensorFlow 的相应函数进行 Dropout
16         if self.dropout_keep_prob < 1:
17             net = tf.nn.dropout(net, keep_prob=self._tf_p_keep)
18         return net
```

其中，上述代码第 6 行处的 `self.use_batch_norm` 与 16 行处的 `self.dropout_keep_prob` 正是我们在增强版神经网络（AdvancedNN）中需要且仅需要引入的模型超参数，因此定义模型超参数

的方法应该继承为：

```
01     def init_model_param_settings(self):
02         # 继承并定义所有在 BasicNN 中定义过的模型超参数
03         super(Advanced, self).init_model_param_settings()
04         # 默认将 Dropout 的 keep_prob 设为 0.5
05         self.dropout_keep_prob = float(self.model_param_settings.get(
06             "keep_prob", 0.5))
07         # 默认对每层的输入都进行规范化处理
08         self.use_batch_norm = self.model_param_settings.get(
09             "use_batch_norm", True)
```

以上就是 Dropout 和 Batch Normalization 的实现。得益于 TensorFlow 的强大，相应的代码还是很简洁的。

5.2 “浅”与“深”的结合

5.1 节叙述的 Dropout 与 BN 旨在解决神经网络的过拟合问题与梯度问题，它们都没有对 BasicNN 本身的结构做出改动，所以我们在本章的开头只将它们称为“技巧（Trick）”。如果想让神经网络拥有更为本质的不同的话，我们还是要从结构入手来探究各种可能性。因此在本节与下一节中，我们将分别介绍两种神经网络的结构设计，它们各有各的优缺点，但都是在 BasicNN 的基础上做了比较本质的创新。对于本节而言，我们将会介绍 Google 发明的一种神经网络结构——Wide and Deep。虽然它的初衷是为了解决推荐系统相关的问题，但是笔者认为，其背后有着一些相当有趣且通用的性质。

注意：本节所将介绍的内容乃笔者在看完 Wide and Deep 原论文后，结合自己的知识与本书的主旨所总结出来的东西，它们有可能有违发明者的本意，也有可能蕴含着一些发明者没有想到的地方。如果对原论文的主旨感兴趣的话，可以直接参见原论文（<https://arxiv.org/pdf/1606.07792.pdf>）。这里关注的重心，还是在如何改进朴素的神经网络上。

5.2.1 离散型特征的处理方式

在正式讨论 Wide and Deep（为了简洁，下文有时会简称它为 WnD）模型之前，我们需要先知道离散型特征的一种处理方式——Embedding。之前在第 2 章讨论传统机器学习算法时，我们使用了 OneHot Encoding 来处理离散型特征，主要是因为传统算法本身比较适应于 OneHot Encoding 这种数据表达形式。而对于神经网络而言，如果离散型特征的特征取值有很多的话，那么 OneHot Encoding 将会是非常不友好的一种处理方法。这是因为，假设某个离散型特征有 n 个取值的话，其 OneHot Encoding 就将会是一个 n 维的向量，且该 n 维向量中只有一维的取值是 1，其余 $n-1$ 维的取值都是 0。这种稀疏性将会使得神经网络中出现很大的冗余，权值矩阵的更新也会因而变得非常缓慢。

所以为了解决这个问题，我们通常会对取值很多的离散型特征（下文统一简称为“稀疏特征（Sparse Features）”）进行 Embedding 的处理。具体而言，假设某个稀疏特征 \mathbf{x} 有 n 个取值，那么我们会维护一个 $n \times k$ 的 Embedding 矩阵：

$$\mathbf{W}_{\text{embed}} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nk} \end{bmatrix}$$

其中，我们认为 $\mathbf{W}_{\text{embed}}$ 的第 i 行即为 \mathbf{x} 的第 i 个取值的“低维连续表示（Low-Dimensional Dense Representation）”。换句话说，我们会建立这样一个一一映射的关系：

$$\mathbf{x}^{(i)} \leftrightarrow \mathbf{W}_{\text{embed}}^{(i)} \triangleq (w_{i1}, w_{i2}, \dots, w_{ik})^T$$

从而当我们将 \mathbf{x} 送进神经网络时，并不是把 \mathbf{x} 的具体取值（ $\mathbf{x}^{(i)}$ ）送进神经网络，而是把该取值对应的低维连续表示（ $\mathbf{W}_{\text{embed}}^{(i)}$ ）送进神经网络。这样对于神经网络而言，离散型特征 \mathbf{x} 就“变成”连续型特征了——因为神经网络每次看到的，都是一个 k 维的、连续的向量。

在这个过程中，需要注意的有如下两点：

- k 的选取。一般而言将 k 全都设置为 8 将会是不错的选择，不过如果有一些特征取值特别多的离散型特征或有一些只有 3 个或 4 个取值的离散型特征的话，那么就可以把 k 设置为取值个数的对数加 1。具体而言，对于有 n 个特征取值的离散型特征 \mathbf{x} 来说，我们可以将 k 取为

$$k = \lceil \log_2 n \rceil + 1$$

其中，“ $\lceil \cdot \rceil$ ”代指“向上取整”，它将会返回比输入大的最小的整数。

- $\mathbf{W}_{\text{embed}}$ 的维护问题。在一些经典的领域，比如自然语言处理领域（Natural Language Processing，常简称为 NLP）中，我们通常利用一些手段来预先获取 $\mathbf{W}_{\text{embed}}$ ，然后在实际训练时就不再维护 $\mathbf{W}_{\text{embed}}$ 。不过对于一般的问题来说，通常的做法是先随机初始化一个 $\mathbf{W}_{\text{embed}}$ ，然后在实际训练时同步训练 $\mathbf{W}_{\text{embed}}$ ，以期望神经网络能够学到一个足够好的、能够反映离散型特征各个取值之间的联系 $\mathbf{W}_{\text{embed}}$ 。

而对于实现来说，因为 TensorFlow 同样有相应的封装，所以相应的代码是很简单的：

```
01 # 参数 i 代指当前正在获取第 i 个离散型特征的 Embedding
02 # 参数 n 代指该离散型特征的特征取值个数
03 def get_embedding(self, i, n):
04     # 我们将会使用 self.embedding_size 这个属性来记录 k
05     # 当它是字符串“log”时，就按照上述的对数公式来计算 k
06     if self.embedding_size == "log":
07         embedding_size = math.ceil(math.log2(n)) + 1
08     # 否则，直接进行相应的赋值即可
09     else:
10         embedding_size = self.embedding_size
11     # 利用“截断正态分布”来初始化  $\mathbf{W}_{\text{embed}}$ ，在此我们将
12     # 初始化均值、标准差分别设为了（默认的）0 和 0.02
13     embedding = tf.Variable(tf.truncated_normal(
```

```

14         [n, embedding size], mean=0, stddev=0.02
15     ), name="Embedding{}".format(i))
16     # 利用 tf.nn.embedding_lookup 来完成
17     # 离散型特征到 Embedding 的映射
18     # 这里用到的 self.categorical_xs, 是储存着所有
19     # 离散型特征对应的 placeholder 的列表
20     return tf.nn.embedding_lookup(
21         embedding, self.categorical_xs[i],
22         name="Embedded X{}".format(i)
23     )

```

其中，上述代码第 21 行中用到的 `self.categorical_xs` 的定义会在本节的最后一小节给出，这里暂时按下不表。

5.2.2 Wide and Deep 模型概述

Wide and Deep 模型的思想其实很简单直观：将“浅（Wide）”模型与“深（Deep）”模型进行结合，从而期望它们能够产生互补的作用，如图 5.3 所示，该图是论文中的原图。

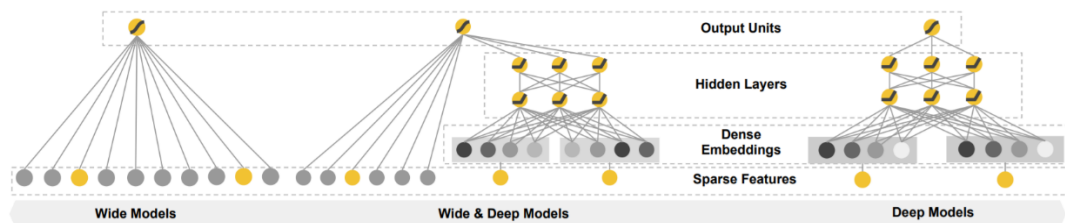


图5.3 WnD模型（1）

其中，所谓的 Wide 模型，就是指没有隐藏层的神经网络，即输入层的特征将会直接被输出层看到。这意味着在 Wide 模型中，特征之间的相互作用将会很容易被“记住”，因为特征经过一个线性映射之后就直接到了输出层并送进损失函数，反向传播时梯度也能直接到达连接着输入层的权值矩阵，从而参数的更新是直接与各特征相联系的。但是，由于此时 Wide 模型毕竟只是一个线性模型，所以它的表达能力是不足的，即虽然它能够记住不少特征之间的作用，但这种记忆是比较“浅薄”的。对于更深层次关系的挖掘，Wide 模型会显得有些力不从心，此时我们就需要用到 Deep 模型了。而所谓的 Deep 模型，其实就只是一个朴素的神经网络（BasicNN）。这是因为 BasicNN 的拟合能力足够强，所以只要特征之间确实有更深层次的关系，那么 BasicNN 从理论上来说就能够找到它，只不过到底能不能通过训练来得到它是一个未知数而已。同时，注意我们在 3.6 节中评估 BasicNN 时，是默认输入数据已经经过数据预处理的。在这里，我们就要针对离散型特征做一个稍微详细的讨论了。

注意：对于连续型特征的处理手段我们会在第 6 章中进行说明，所以这里仍然认为连续型特征都已经做好了数据预处理。

首先是 Wide 模型（线性模型）的输入，原论文中表示它应该由如下两部分组成：

- 输入数据的原始特征。

- 由原始特征推算出来的“转换特征（Transformed Features）”。

对于转换特征的获取，原论文给出了一种叫 Cross Product Transformation 的算法。具体而言，假设原始特征向量为

$$\mathbf{x} = (x_1, x_2, \dots, x_d)^T$$

那么该算法就会通过

$$\tilde{x}_k \triangleq \prod_{i=1}^d x_i^{c_{ki}}$$

来获取第 k 个转换特征，其中 $c_{k1}, c_{k2}, \dots, c_{kd}$ 是 k 个 0-1 变量，它们只能在 0 和 1 这两个数里面取值。换句话说，该算法的本质其实只是挑选出原始特征向量中某些维度的特征，然后进行一个简单的相乘而已。这种算法对应的转换特征在 $x_1 \sim x_d$ 都是二值特征时，其实就是著名的“交叉特征”。具体而言，假设 $x_1 \sim x_d$ 满足 $x_1 \sim x_d \in \{0,1\}$ ，且只挑选 $x_1 \sim x_2$ 来计算转换特征 \tilde{x}_k 的话，则

$$\tilde{x}_k = \prod_{i=1}^d x_i^{c_{ki}} = x_1 x_2 \in \{0,1\}$$

即 \tilde{x}_k 也将会是一个二值特征。此时就会发现，如果借鉴第 2 章讨论朴素贝叶斯时对二值特征的理解，并认为当二值特征取 1 时意味着该特征“出现了”、取 0 则意味着“没出现”的话，那么转换特征 \tilde{x}_k 就只会在 $x_1 \sim x_2$ 同时在原始特征向量中“出现”（即 $x_1 = x_2 = 1$ ）时“出现”（即此时才会有 $\tilde{x}_k = 1$ ），在其余情况下都不会出现，从而就蕴含了交叉特征的性质。此外，同样由第 2 章朴素贝叶斯处的讨论可知，对于一般的、有 n 个特征取值的离散型特征，我们总可以把它“展开”成 n 个二值特征，所以上述算法得到的转换特征对于一般的离散型特征而言，也可以视为交叉特征。

不过，该算法中有一个非常不优雅的地方，它需要我们手动去构造 $c_{k1} \sim c_{kd}$ ，这就意味着该算法需要人力的介入而不能让机器自己去完成整个训练流程，从而模型的表现将会在很大程度上取决于我们先验知识的有效性，这无疑就给后续的调整带来了很大的麻烦。因此在下一节中，我们将介绍一种能够在一定程度上自动化计算转换特征过程的结构；在这一节介绍具体实现时，我们就暂时先假设转换特征的获取不成问题。

接下来说说 Deep 模型（朴素神经网络）的输入。正如上一节所说的，当某个离散型特征的特征取值过多时，朴素神经网络就会有点吃不消它的 OneHot Encoding，此时就需要用到 Embedding 技术。所以一般而言，我们会把 Deep 模型的输入设置为连续型特征与做了 Embedding 后的离散型特征的“合并（Concat）”。

原论文中有一幅图很好地说明了如何将连续型特征与做了 Embedding 后的离散型特征进行合并，同时该图也说明了如何将 Wide 模型与 Deep 模型进行合并，如图 5.4 所示。

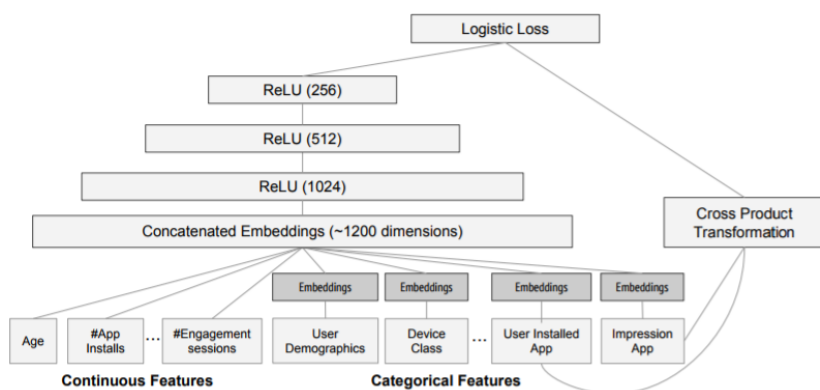


图 5.4 WnD模型（2）

可以看到在图 5.4 中，Wide 模型（图中最右的部分）接收的输入只有“User Installed App”和“Impression App”通过 Cross Product Transformation 算法得到的转换特征，而 Deep 模型（图中的主体部分）接收的输入则是所有连续型特征和所有离散型特征做了 Embedding 后的特征的合并。

5.2.3 Wide and Deep 的具体实现

前两节我们从理论上说明了 Wide and Deep 的思想与结构，这一节我们就要具体地把它实现出来了。由前文的讨论不难看出，Wide and Deep 的模型搭建本身不算困难——只需把神经网络和线性模型（即退化的神经网络）分别写一遍，然后把得到的两个原始输出加起来视为最终的原始输出即可（原论文没有显式地指出这一点，但笔者通过查阅 TensorFlow 源代码之后发现 Google 确实是这样实现的）。Wide and Deep 真正的困难之处，其实在于模型输入的准备与管理上。

由 Wide and Deep 的特性可知，我们需要知道在给进来的原始数据中，哪些维度的特征是离散型特征、哪些维度的特征是连续型特征，所以一个自然的想法就是用一个列表来储存各个特征的类型。具体而言，我们会在新的神经网络模型里面定义一个叫 `self.numerical_idx` 的属性，它储存的元素为 True 或者 False，其中：

- 第 i 个元素是 True，则意味着第 i 个特征是连续型特征。
- 第 i 个元素是 False，则意味着第 i 个特征是离散型特征。

此外，`self.numerical_idx` 的最后一个元素将会是标签的类型。具体而言：

- `self.numerical_idx[-1]` 为 True 意味着标签是连续的，从而目标就是解决回归问题。
- `self.numerical_idx[-1]` 为 False 意味着标签是离散的，从而目标就是解决分类问题。

除了 `self.numerical_idx` 以外，由于我们在定义离散型特征的 Embedding 时需要用到该离散型特征的特征取值个数，所以我们还要用另外一个列表来储存该信息。具体而言，我们会定义一个叫 `self.categorical_columns` 的属性，它储存的元素是一个元组 (i, n) ，其中：

- i 意味着第 i 个特征是离散型特征。

- n 意味着该离散型特征有 n 个取值。

下面就举一个小例子来辅助理解这两个属性。假设我们现在有 5 个特征，其中第 0、3、4 个特征是离散的，它们分别有 2、8、10 个取值，而且问题是回归问题的话，就有：

```
self.numerical_idx = [False, True, True, False, False, True]
```

以及

```
self.categorical_columns = [(0,2), (3,8), (4,10)]
```

注意：不难看出，`self.categorical_columns` 的信息是包含了 `self.numerical_idx` 的。之所以用了两个属性，是因为这样在实现其他功能的时候会更方便一些。

同时，延续之前的实现规范，我们会使用字典（`self.data_info`）来存储像 `self.numerical_idx` 和 `self.categorical_columns` 这样的与数据信息相关的超参数（后文统一简称为“数据超参数”）。此外，我们在 5.2.1 节最后展示 Embedding 实现时，还用到了 `self._categorical_xs` 这个属性。这个属性对应的是一个列表，其每个元素都是原始特征里面某个离散型特征所对应的 placeholder。当实际训练模型时，需要把相应的离散型特征的具体取值 feed 给这些 placeholder，而这一步也正是 Wide and Deep 的输入难以管理的根本原因之一。

那么在知道了大致的管理思路后，我们就能来看具体的实现了。首先来看看这些属性是如何进行初始化的。考虑到代码展示的连贯性，我们直接在此展示完整的、增强版神经网络（AdvancedNN）的初始化（完整的代码可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/e_AdvancedNN/NN.py），如代码 5.1 所示。

代码 5.1 AdvancedNN 的初始化：e_AdvancedNN/NN.py

```
01 # 继承朴素神经网络 BasicNN 并在其基础上进行拓展
02 class Advanced(Basic):
03     # 定义模型的“签名”
04     signature = "Advanced"
05
06     """
07     初始化结构
08     name: 模型的名字
09     data info: 管理“数据超参数”的字典
10     model param settings: 管理“模型超参数”的字典
11     model structure settings: 管理“结构超参数”的字典
12     """
13     def init (self, name=None, data_info=None,
14               model_param_settings=None, model_structure_settings=None):
15         # 定义一个新的、用于保存模型的属性
16         # 该属性的具体用法会在后文给出，这里暂时按下不表
17         self.tf_list_collections = None
18         # 调用之前的初始化方法，完成继承
19         super(Advanced, self). init (
20             name, model_param_settings, model_structure_settings)
```



```

21     self.name.append("AdvancedNN")
22
23     # 初始化“数据超参数”
24     if data_info is None:
25         self.data_info = {}
26     else:
27         assert msg = "data info should be a dictionary"
28         assert isinstance(data_info, dict), assert msg
29         self.data_info = data_info
30     # 用 self.data_info.initialized 来标识是否已经初始化过数据超参数
31     self.data_info.initialized = False
32     # 定义出两个核心数据超参数
33     self.numerical_idx = self.categorical_columns = None
34
35     # 定义一些 WnD 输入相关的属性
36     # self.deep_input、self.wide_input 分别代指
37     # Deep 模型与 Wide 模型的输入的 Tensor
38     self.deep_input = self.wide_input = None
39     self.categorical_xs = None
40     self.embedding_size = None
41     # 由于 Wide 模型的 Cross Product Transformation 可能会
42     # 用到 OneHot Encoding、Deep 模型则可能会用到 Embedding,
43     # 所以我们要定义许多属性来代指这些属性相关的 Tensor
44     # 这些 Tensor 的具体内涵马上在后文给出, 此处暂时按下不表
45     self.embedding = self.one_hot = None
46     self.embedding_concat = self.one_hot_concat = None
47     self.embedding_with_one_hot = None
48     self.embedding_with_one_hot_concat = None
49
50     # 这两个属性就是在本章第 1 节用到的两个属性
51     self.dropout_keep_prob = self.use_batch_norm = None
52     # 用于定义模型结构的三个属性, 其中
53     # 1) self.use_wide_network 用于指定是否使用 Wide 模型
54     # 如果该属性为 False, 那么 WnD 就会“退化”成 BasicNN
55     # 2) self._dndf 代指 DNDF 结构
56     # DNDF 会在下一节中进行说明, 这里暂时按下不表
57     # 3) self._pruner、self._dndf_pruner 分别代指在
58     # 神经网络中与 DNDF 中运用剪枝技术的具体实例
59     # 剪枝技术会在 5.4 节里进行说明, 这里暂时按下不表
60     self._use_wide_network = self._dndf = None
61     self._pruner = self._dndf_pruner = None
62
63     # 定义额外可能用到的 placeholder, 其中
64     # 1) self._tf_p_keep 已在 5.1 节中用到了, 代指 Dropout 的 keep_prob
65     # 2) self._n_batch_placeholder 将会在下一节里用到
66     self._tf_p_keep = None
67     self._n_batch_placeholder = None
68
69     # 定义代指“真正有效的 numerical_idx”的 property

```

```

70     # 它会返回 numerical idx 中的非 None 元素
71     # 其具体功用在下一章才会显现出来
72     @property
73     def valid_numerical_idx(self):
74         return np.array([
75             is_numerical for is_numerical in self.numerical_idx
76             if is_numerical is not None
77         ])

```

需要指出的是，我们在代码的一开始（第 19 行）调用了“super”这个 Python 的内置方法。super 的具体说明可以参见官方文档 (<https://docs.python.org/3.6/library/functions.html#super>)，而对于本书而言，我们只需知道它能帮我们把继承对象的相应方法调用一遍即可。具体而言，在 AdvancedNN 看来，由于它继承的是 BasicNN，所以一旦它用到了 super，就意味着它会把 BasicNN 的相应方法执行一遍。

总之，现在我们知道了 AdvancedNN 各个属性的意义，所以接下来只需把它们逐一定义出来即可。首先是数据超参数的初始化：

```

01     def init_data_info(self):
02         # 如果已经初始化过数据超参数，就什么都不做
03         if self.data_info_initialized:
04             return
05         # 否则就进行数据超参数的初始化
06         self.data_info_initialized = True
07         self.numerical_idx = self.data_info.get("numerical_idx", None)
08         self.categorical_columns = self.data_info.get("categorical_columns", None)
09         if self.numerical_idx is None:
10             raise ValueError("numerical_idx should be provided")
11         if self.categorical_columns is None:
12             raise ValueError("categorical_columns should be provided")

```

由上述代码可以看出，对于 AdvancedNN 而言，我们要求调用者必须提供 self.numerical_idx 和 self.categorical_columns 的具体取值。至于自动获取这两个属性的取值的方法，我们会在附录 F 中进行相应的说明。

然后就是利用数据来具体初始化各个参数的方法，该方法倒是没有原来的基础上做出太多的变动：

```

01     def init_from_data(self, x, y, x_test, y_test, sample_weights, names):
02         # 先初始化数据超参数，再利用数据来进行具体的初始化
03         self.init_data_info()
04         super(Advanced, self).init_from_data(
05             x, y, x_test, y_test, sample_weights, names)
06         # 处理特征维度。由于在 BasicNN 中，我们其实默认了所有特征
07         # 都是连续型特征，所以这里我们要减去离散型特征的数目
08         # 从而获得真正的连续型特征的个数
09         self.n_dim -= len(self.categorical_columns)

```

不过在真正定义模型输入与各种 placeholder 的部分，Wide and Deep 要比 BasicNN 复杂多

了，为此我们会分两步来对相应的实现进行说明。

首先是前半段的实现。在展示具体的代码之前，为了便于理解，我们先来梳理一下即将用到或实现的 `placeholder` 与 `Tensor`。

- `self._tfx`: 输入特征向量中的连续型特征。注意在第3章介绍 `BasicNN` 的实现时，该属性代指的 `placeholder` 直接就是模型输入中特征向量对应的 `placeholder`，这是因为当时我们认为模型只有连续型特征的输入，而这一章我们将会考虑离散型特征。所以接下来的实现中需要特别注意的是，`self._tfx` 对应的其实是连续型特征而不一定是模型真正看到的特征。
- `self._categorical_xs`: 储存各个离散型特征所对应的 `placeholder` 的属性。
- `self._one_hot`: 离散型特征的 OneHot Encoding（以下简称为 OneHot）。
- `self._embedding`: 离散型特征的 Embedding（以下简称为 Embedding）。
- `self._embedding_with_one_hot`: OneHot 与 Embedding 的合并。
- `self._one_hot_concat`: 连续型特征（如果有）+ OneHot。
- `self._embedding_concat`: 连续型特征（如果有）+ Embedding。
- `self._embedding_with_one_hot_concat`: 连续型特征（如果有）+ OneHot + Embedding。

在知道了这些属性的具体内涵后，相应的实现虽然很长，但其脉络就比较清晰了。

```

01 def _define_input_and_placeholder(self):
02     # 继承 BasicNN 的相应方法
03     super(Advanced, self)._define_input_and_placeholder()
04     # 如果没有离散型特征的话，那么上述所有 Tensor 都
05     # 直接用 self._tfx（连续型特征）赋值即可
06     if not self.categorical_columns:
07         self.categorical_xs = []
08         self.one_hot = self.one_hot_concat = self._tfx
09         self.embedding = self.embedding_concat = self._tfx
10         self.embedding_with_one_hot = self._tfx
11         self.embedding_with_one_hot_concat = self._tfx
12     # 否则，我们就要逐一建立 placeholder 并进行后续的
13     # OneHot Encoding 或 Embedding 操作
14     else:
15         # 如果连续型特征个数（self.n_dim）为 0，
16         # 就意味着都是离散型特征
17         all_categorical = self.n_dim == 0
18         with tf.name_scope("Categorical Xs"):
19             # 定义出 self._categorical_xs
20             self.categorical_xs = [
21                 tf.placeholder(tf.int32, shape=[None],
22                               name="Categorical X{}".format(i))
23                 for i in range(len(self.categorical_columns))
24             ]
25         with tf.name_scope("One hot"):
26             # 利用特征取值个数与 tf.one_hot 直接获取 OneHot Encoding
27             one_hot_vars = [

```

```

28         tf.one_hot(self._categorical_xs[i], n)
29         for i, ( , n) in enumerate(self.categorical_columns)
30     ]
31     self.one_hot = tf.concat(one_hot_vars, 1, name="Raw")
32     # 如果有连续型特征的话
33     # 就生成一个“连续型特征+OneHot”的输入 Tensor
34     if not all_categorical:
35         self.one_hot_concat = tf.concat(
36             [self._tfx, self._one_hot], 1, name="Concat")
37     else:
38         self._one_hot_concat = self._one_hot
39     with tf.name_scope("Embedding"):
40         # 利用特征取值个数与 5.2.1 节的相应实现获取 Embedding
41         embeddings = [
42             self._get_embedding(i, n)
43             for i, ( , n) in enumerate(self.categorical_columns)
44         ]
45         self._embedding = tf.concat(embeddings, 1, name="Raw")
46         # 如果有连续型特征的话
47         # 就生成一个“连续型特征+Embedding”的输入 Tensor
48         if not all_categorical:
49             self._embedding_concat = tf.concat(
50                 [self._tfx, self._embedding], 1, name="Concat")
51         else:
52             self._embedding_concat = self._embedding
53     # 利用 concat 把 OneHot 和 Embedding 进行“合并”
54     with tf.name_scope("Embedding_with_one_hot"):
55         eo = self._embedding_with_one_hot = tf.concat(
56             embeddings + one_hot_vars, 1, name="Raw")
57         # 如果有连续型特征的话
58         # 就生成“连续型特征+Embedding+OneHot”的输入 Tensor
59         if not all_categorical:
60             self._embedding_with_one_hot_concat = tf.concat(
61                 [self._tfx, self._embedding_with_one_hot],
62                 1, name="Concat")
63         else:
64             self._embedding_with_one_hot_concat = eo

```

以上就是前半段的代码实现，下面来看看后半段的代码。同样的，为了便于理解，我们先梳理一下将要用到或实现的 placeholder 和 Tensor。

- `self._wide_input`: Wide 模型的输入对应的 placeholder。
- `self._deep_input`: Deep 模型的输入对应的 placeholder。
- `self._tf_p_keep`: Dropout 中将会用到的 Tensor。当 `self._is_training` 为 True（即模型正在训练）时，它就应该取值为 Dropout 的 `keep_prob`；当 `self._is_training` 为 False（即模型没在训练而在预测）时，它就应该取值为 1（即不进行 Dropout）。
- `self._n_batch_placeholder`: 代指当前 Mini Batch 大小的 placeholder。我们将在下一节中用到它，这里暂时按下不表。

此外，为了让用户指定模型输入时更方便，我们会利用 `getattr` 方法来让用户以字符串的形式来分别指定 Wide 模型与 Deep 模型的输入。综上所述，后半段的代码实现就比较清晰、直观了。

```

65     # 如果用户只希望 Wide 模型接收连续型特征 (continuous)
66     # 就将 self. tfx 赋给 self. wide input
67     if self. wide input == "continuous":
68         self. wide input = self. tfx
69     # 否则，利用 getattr 方法来给 self. wide input 赋值
70     # 比如，如果用户想让 Wide 模型接收连续型特征+Embedding 的话
71     # 就只需将 self. wide input 初始化为 embedding concat 即可
72     else:
73         self. wide input = getattr(self, " " + self. wide input)
74     # Deep 模型的输入与 Wide 模型完全同理，所以就不重复给出注释了
75     if self. deep input == "continuous":
76         self. deep input = self. tfx
77     else:
78         self. deep input = getattr(self, " " + self. deep input)
79     # 利用 self. define hidden units 方法来初始化隐藏层信息
80     if self.hidden units is None:
81         self. define hidden units()
82     # 利用 tf.cond 来在计算图中写入条件语句
83     self. tf p keep = tf.cond(
84         self. is training, lambda: self.dropout keep prob, lambda: 1.,
85         name="keep prob"
86     )
87     # 定义出 self. n batch placeholder
88     self. n_batch_placeholder = tf.placeholder(tf.int32, name="n_batch")

```

然后后半段的代码不长，但是却有许多需要补充说明的地方。首先是 `self_wide_input` 和 `self_deep_input` 的初始化问题，由上述代码与上文的说明可知，它们是需要被初始化成字符串的。而由于它们都属于结构超参数，而 5.3 节和 5.4 节将要介绍的 DNDF 与剪枝技术也都会引入结构超参数，所以我们会在 5.5 节介绍这些技术的互补性时补充展示初始化结构超参数的代码实现。

然后是 `self_define_hidden_units` 这个方法。由于本章主要介绍的 AdvancedNN 拥有比较强的拟合能力与泛化能力，所以我们可以只根据样本的数量来经验性地定出比较合适的隐藏层层数以及各个隐藏层中的神经元个数。具体的实现只是一些 `if`、`else` 的堆叠，直接贴过来不免有些冗余，所以这里就只提供一个链接，感兴趣的读者可以去看看：https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/e_AdvancedNN/NN.py。

最后是 TensorFlow 中 `tf.cond` 这个函数。我们在 3.5.1 节中介绍 TensorFlow 的组成单元时曾经提到过它，然后正如上述代码的第 82 行所说，我们能够利用它在 TensorFlow 的计算图 Graph 中写入条件语句（Conditional Statement）。具体而言，假设我们现在有一个叫 `cond` 的、数据类型为 `tf.bool`（即只能取 `True` 和 `False` 这两个值）的 `placeholder`，然后我们希望构造一个叫 `state`

的 Tensor, 如果它能在 `cond` 为 `True` 时取值为函数 f_1 的返回值、在 `cond` 为 `False` 时取值为函数 f_2 的返回值的话, 那么 `state` 就可以这样来定义:

```
state = tf.cond(cond, f1, f2)
```

回到 `self_tf_p_keep` 的实现来看的话, 由于 `self_is_training` 是代指模型是否正处于训练过程的 `placeholder`, 所以上述代码的第 83~86 行的效果就是:

- 当模型正在训练时, `self_tf_p_keep` 就会取值为函数 “`lambda: self.dropout_keep_prob`” 的返回值。这里的 `lambda` 是 Python 中的匿名函数, 如果对其不熟悉的话, 可以参见官方文档 (<https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions>)。而如果知道它的用法的话就能看出, 该匿名函数将总是返回 `self.dropout_keep_prob` 这个属性。由于该属性就是 Dropout 中的 `keep_prob`, 所以当模型处于训练过程中时, `self_tf_p_keep` 确实就会取值为 `keep_prob`。
- 当模型没在训练时, `self_tf_p_keep` 就会取值为函数 “`lambda: 1`” 的返回值, 即确实就会取值为 1。

以上就是所有模型输入的相关实现了, 内容确实挺多, 但总体的思路还是比较清晰直观的。而至于模型本身的搭建, 则不是特别困难的事。

```
01 # 定义搭建 Wide and Deep 模型的方法
02 def build_model(self, net=None):
03     # 先继承 BasicNN 的方法, 把 Deep 模型搭建完毕
04     # 这里我们把 self._deep_input 传给了 BasicNN 的搭建方法
05     # 意味着我们指定了 BasicNN 的输入为 self._deep_input
06     super(Advanced, self)._build_model(self._deep_input)
07     # 如果指定了要使用 Wide 模型的话, 就进行相应的搭建
08     if self.use_wide_network:
09         # 如果 self._dndf 是 None, 就意味着 Wide 模型只是一个
10         # 普通的线性模型而已, 从而调用 3.5.2 节中定义过的
11         # self._fully_connected_linear 方法来完成搭建即可
12         if self._dndf is None:
13             wide_output = self._fully_connected_linear(
14                 self.wide_input, appendix="wide output",
15                 shape=[self.wide_input.shape[1].value, self.n_class]
16             )
17         # 否则, 利用 5.3 节将介绍的 DNDF 结构来搭建 Wide 模型
18         else:
19             wide_output = self._dndf(self.wide_input, self.n_batch_placeholder)
20         # 模型的最终输出就是将 Deep 模型和 Wide 模型的输出加起来
21         self._output += wide_output
```

以上就是 Wide and Deep 模型的全部实现。由于我们后续要介绍的 DNDF 结构与剪枝技术都将会在 WnD 的基础上应用, 所以这些实现也正是 AdvancedNN 主体框架的实现。不难看出我们引入了许多关键的属性, 因此如果想把它保存下来的话, 需要在 BasicNN 的保存方法的基础上进行一些拓展。

```

01     # 由于我们新引入了“数据超参数”，所以需要额外地把
02     # self.data_info 这个属性保存下来
03     def define_py_collections(self):
04         super(Advanced, self).define_py_collections()
05         self.py_collections.append("data_info")
06
07     # 由于定义了大量与模型输入相关的 Tensor，所以需要
08     # 额外地把这些 Tensor 和 placeholder 对应的指针都保存下来
09     def define_tf_collections(self):
10         super(Advanced, self).define_tf_collections()
11         self.tf_collections += [
12             "deep input", "wide input", "n batch placeholder",
13             "embedding", "one hot", "embedding with one hot",
14             "embedding concat", "one hot concat",
15             "embedding with one hot concat"
16         ]
17     # 此外，还需要对 self.categorical_xs 这个属性做特殊处理
18     self.tf_list_collections = ["_categorical_xs"]

```

其中，在代码的第 18 行我们用到了一个全新的属性——`self.tf_list_collections`。这主要是因为 `self.categorical_xs` 这个属性是存储所有离散型特征对应的 `placeholder` 的列表的，而第 3 章所介绍的保存指向 `Tensor` 和 `placeholder` 的指针的 `tf.add_to_collection` 函数，是不能用来保存列表的，因此我们就需要把 `self.categorical_xs` 里面的元素逐一进行保存。为了方便拓展，我们统一把这种存储了一系列 `Tensor` 和 `placeholder` 的指针的列表放进 `self.tf_list_collections` 中进行管理，于是第 3 章中介绍过的保存、复用与清空指针的方法就需要拓展为：

```

01     def add_tf_collections(self):
02         super(Advanced, self).add_tf_collections()
03         # 对于 self.tf_list_collections 中的所有列表，将其中的
04         # Tensor 或 placeholder 的指针进行保存
05         for tf_list in self.tf_list_collections:
06             target_list = getattr(self, tf_list)
07             if target_list is None:
08                 continue
09             for tensor in target_list:
10                 tf.add_to_collection(tf_list, tensor)
11
12     def restore_collections(self, folder):
13         # 利用 setattr 来给 self.tf_list_collections 中的所有属性赋值
14         # 注意，由于 tf.get_collection 返回的是一个 list，所以直接赋值即可
15         for tf_list in self.tf_list_collections:
16             if tf_list is not None:
17                 setattr(self, tf_list, tf.get_collection(tf_list))
18         # 由于 BasicNN 的 restore_collections 方法中蕴含了
19         # 清空指针的逻辑，所以要放在最后
20         super(Advanced, self).restore_collections(folder)
21
22     def clear_tf_collections(self):

```

```
23         super(Advanced, self).clear_tf_collections()
24         # 注意要将 self.tf_list_collections 中的指针也清空
25         for key in self.tf_list_collections:
26             tf.get_collection_ref(key).clear()
```

以上就是 Wide and Deep 模型的保存与复用的全部实现。由于在 BasicNN 处已经将大部分工作都做好了，所以在此基础上进行拓展是比较方便、简单的。

此外，如果想要真正训练并应用 Wide and Deep 模型的话，我们还需要额外拓展构造模型训练时、其输入所涉及的 placeholder 所需的 feed_dict 的 self.get_feed_dict 方法。不过与初始化结构超参数类似，后续的 DNDF 结构与剪枝技术都需要对该方法做出修改，所以我们会统一在 5.5 节进行相应的代码展示。

5.2.4 WnD 的重要思想与优缺点

前三节大致说明了一下 Wide and Deep 的结构与相应实现，不过对于为何要使用 WnD 还没有进行过讨论。在笔者看来，WnD 最重要的思想就在于“浅”与“深”的结合上。从直观上来说，对于一个简单的数据集而言，浅的模型就已经足够训练出很好的结果了，此时如果非得用很深的模型去拟合它，那么最终反而可能会遇到过拟合的问题。事实上，3.6 节最后用到的噪声数据集就是非常典型的例子，浅模型（SVM）的表现要远比深模型（BasicNN）的表现好。但浅模型毕竟无法适应复杂的情况，所以深模型又是不可或缺的。一种经常会陷入的思维陷阱是，每次都先试一试，然后决定用浅模型还是深模型。但事实上，我们完全可以将它们结合起来，并期望能同时适应于简单情况与复杂情况。

然而，虽然这种想法是自然且美好的，但实际操作起来却没那么简单。Wide and Deep 虽然不失为一种好的解决方案，但它（直接将 Wide 模型与 Deep 模型的输出加在一起作为最终输出的做法）还是有些过于简单粗暴了。而事实上，这种太过直接的方式会带来一些严重的问题，比如 Wide 模型的影响通常会过于强大，以至于盖过 Deep 模型的影响。换句话说，虽然我们期望 WnD 能够自动地在浅与深之间找到一个平衡点，但由于 Wide 模型本身很容易训练，且其输入和输出之间是直接通过线性映射相连接的，所以 Wide 模型就经常会占据“主导地位”。这对于通常会比较复杂的现实任务而言是不利的，因为 Wide 模型只能“记住”浅层的关系而无法像 Deep 模型那样挖掘到深层的信息。

总之，虽然仍然存在一些问题，但 Wide and Deep 这种“结合浅与深”的思想还是非常重要的。由于前文的诸多讨论也反映了 WnD 的方方面面，所以为了帮助大家回忆本节所讲过的主要内容与核心思想，在本节的最后，我们就来结合前文的诸多讨论并对 WnD 的优缺点进行一个精简的总结。

优点：

- 能够很好地结合离散型特征和连续型特征。
- 在简单模型和复杂模型的选择这个问题上有天然的适应性。

缺点：

- 需要手动去选取一些交叉特征。
- 容易倾向于训练简单模型（Wide 模型）。

5.3 神经网络中的“决策树”

5.2 节我们介绍了 Wide and Deep 模型，该模型有不少振奋人心的性质，但是却和神经网络中的 Universal Approximation Theorem 一样显得有些触不可及。而这一节所要讲的结构——Deep Neural Decision Forest（常简称为 DNDF），其实就能使得 WnD 的优良性质变得不再那么遥远。不过正如本章开头所说，我们会在 5.5 节介绍这些技术之间的互补性，所以对于本节而言，我们会把注意力集中在 DNDF 本身上面。

5.3.1 DNDF 结构概述

我们在第 3 章曾经介绍过如何把决策树转换成神经网络，当时也指出了这种转换的局限性：本身的人为操弄痕迹太重，以及无法适应于集成算法等。我们在这一节中介绍的 DNDF 结构，比较完美地解决了这两个问题，它既非常自然地在神经网络中引入了树的结构，又天生适应于集成算法。

提出 DNDF 这个结构的论文是 2015 年 ICCV（计算机视觉领域中最高级别的会议）的 Marr Prize Paper（即所谓的 Best Paper），作者是 Peter Kotschieder 等人。虽然该结构的初衷是解决视觉方面的问题，但是正如 Wide and Deep 的初衷是解决推荐系统的问题一样，它本身是一项非常有用的技术，我们不必因为它的初衷而限制了它的应用。

注意：所以类似的，本节将叙述的内容同样可能是有违发明者本意或发明者本身也没想到的。如果对原论文感兴趣的话，可以参见 https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Kotschieder_Deep_Neural_Decision_ICCV_2015_paper.pdf，我们主要还是关心它如何在 BasicNN 的基础上做出改进。

由于 DNDF 结构是“对标”决策树的，所以自然应该拿它和决策树来做对比。为此，下面将会列出决策树的缺点以及 DNDF 相应的解决方案，这些陈列将会是后文叙述 DNDF 结构的一个线索，记住它们或至少对它们有印象对于理解 DNDF 而言是比较重要的。

先来看看决策树的两大缺陷，它们或多或少都在第 2 章和第 4 章有所提及。

- 决策边界是硬边界。所谓的硬边界，就是指决策树中“非黑即白”的分类准则。以二叉树为例，样本在某个中间节点处只能要么往左子节点走，要么往右子节点走，不能说有 80% 的概率往左走、20% 的概率往右走。这种分类准则通常被称为“二分路由”，因为它只有两种选择。
- 决策面通常垂直于坐标轴，这一点在第 4 章曾反复地强调过。

而在 DNDF 中，这两大缺陷都被相当漂亮地解决了。

- DNDF 采用了“概率路由（Stochastic Routing）”，即它允许让一个样本“有 80% 的概率往左走、20% 的概率往右走”，从而直接规避了二分路由带来的硬边界问题。
- DNDF 在从特征层过渡到树结构时采用的是全连接的连接形式，所以决策面可以是任意的超平面。

在知道了 DNDF 的优良性质后，下面我们就来具体地叙述其结构并以此解释为什么它能有这些性质。虽然 DNDF 的原论文看上去比较复杂，不过如果直观来说的话，它的本质其实就是用神经网络中的神经元来表示一棵完全二叉树的中间节点，然后通过一系列的运算模拟出决策树的功能和性质。原论文中有一张很好的图能帮助我们理解这一点，如图 5.5 所示。为了方便大家看清楚具体的符号，笔者修了一下原论文的彩图。

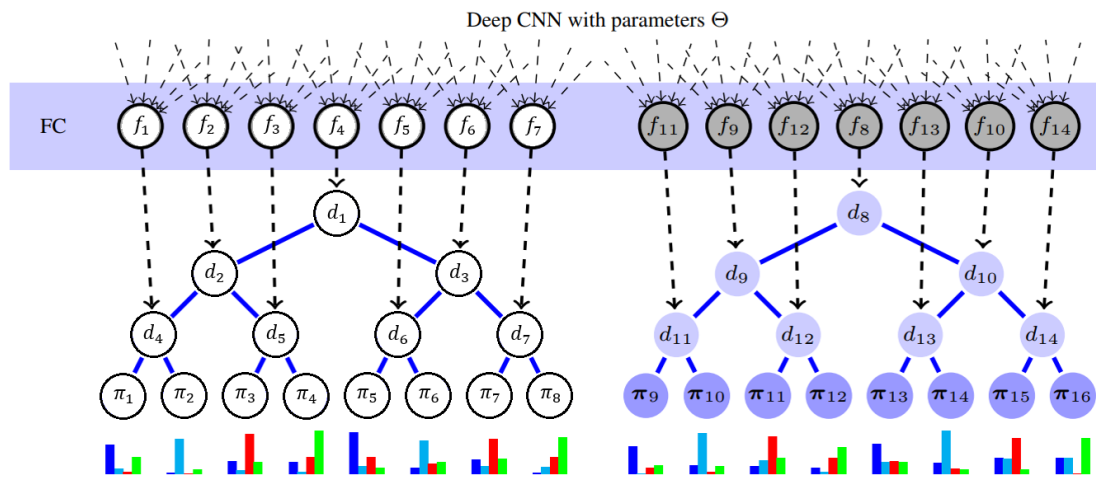


图5.5 DNDF结构

从图 5.5 中可以看出，完整的 DNDF 结构会含有很多棵完全二叉树（虽然图中的 DNDF 只含有 2 棵，但一般默认设置为 10 棵或 16 棵，这也是为何它的名字中带有 Forest 的原因），然后它会将某个 FC（Fully Connected，全连接）层的神经元（ f_1, f_2, \dots ）一一映射到这些完全二叉树的中间节点（ d_1, d_2, \dots ）上。而且原论文还指出，这种映射关系通常取为恒同映射即可：

$$d_i \triangleq f_i, \quad \forall i$$

注意：需要特别指出的是，这里的 FC 层不是特征层，而是特征层后的全连接层，我们一般会称之为节点层。

而在每棵完全二叉树的内部，DNDF 是通过概率路由来划分样本的，这与决策树的二分路由有着较为本质的不同。具体而言，DNDF 认为当样本在中间节点 d_i 处时，就会“有 $\sigma(d_i)$ 的概率往左走，有 $1 - \sigma(d_i)$ 的概率往右走”。其中， σ 函数就是第 3 章中曾经介绍过的 Sigmoid 函数：

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

由于它的输入总是大于 0 小于 1，所以拿它来定义概率是合理的。而事实上在第 2 章介绍 Logistic 回归时，我们也用了 σ 函数来定义概率。

下面就以图 5.5 中的第 1 棵完全二叉树为例，具体说明一下概率路由的操作方式。比如，样本通过左边的决策路径到达左边的叶节点 (π_1) 的概率即为

$$p(\mathbf{x} \in \pi_1) = \sigma(d_1)\sigma(d_2)\sigma(d_4)$$

而到达左数第二个叶节点 (π_2) 的概率即为

$$p(\mathbf{x} \in \pi_2) = \sigma(d_1)\sigma(d_2)[1 - \sigma(d_4)]$$

为了简洁，我们统一将 $\sigma(d_i)$ 简记为 σ_i 、将 $1 - \sigma(d_i)$ 简记为 $\tilde{\sigma}_i$ 、将 $p(\mathbf{x} \in \pi_j)$ 简记为 p_j ，那么上述两个公式就能写成

$$p_1 = \sigma_1\sigma_2\sigma_4, \quad p_2 = \sigma_1\sigma_2\tilde{\sigma}_4$$

以及不难得出，样本属于剩下各个叶节点的概率分别为

$$p_3 = \sigma_1\tilde{\sigma}_2\sigma_5, \quad p_4 = \sigma_1\tilde{\sigma}_2\tilde{\sigma}_5$$

$$p_5 = \tilde{\sigma}_1\sigma_3\sigma_6, \quad p_6 = \tilde{\sigma}_1\sigma_3\tilde{\sigma}_6$$

$$p_7 = \tilde{\sigma}_1\tilde{\sigma}_3\sigma_7, \quad p_8 = \tilde{\sigma}_1\tilde{\sigma}_3\tilde{\sigma}_7$$

为了方便叙述，这里我们额外定义一个决策概率向量 \mathbf{p} ：

$$\mathbf{p} = (p_1, p_2, \dots, p_8)^T$$

对于一般的情况，决策概率向量即为

$$\mathbf{p} = (p_1, p_2, \dots, p_{2^{n+1}})^T$$

其中 n 是树的深度。需要指出的是，DNDF 的概率路由所计算出来的决策概率向量会拥有一个非常好的性质：在每棵树中，样本属于各个叶节点的概率加起来将会是 1，即

$$\sum_{j=1}^{2^{n+1}} p_j = 1$$

对于图 5.5 所示的 DNDF 结构而言，树的深度是 2（根节点不算进深度中），于是就有

$$\sum_{j=1}^8 p_j = 1$$

由于该结构比较简单，所以我们用直接计算的方式来证明这一点。首先可以计算出

$$\begin{aligned} \sum_{j=1}^4 p_j &= p_1 + p_2 + \sum_{j=3}^4 p_j = \sigma_1\sigma_2\sigma_4 + \sigma_1\sigma_2\tilde{\sigma}_4 + \sum_{j=3}^4 p_j \\ &= \sigma_1\sigma_2(\sigma_4 + \tilde{\sigma}_4) + p_3 + p_4 = \sigma_1\sigma_2 + \sigma_1\tilde{\sigma}_2\sigma_5 + \sigma_1\tilde{\sigma}_2\tilde{\sigma}_5 \\ &= \sigma_1\sigma_2 + \sigma_1\tilde{\sigma}_2(\sigma_5 + \tilde{\sigma}_5) = \sigma_1\sigma_2 + \sigma_1\tilde{\sigma}_2 = \sigma_1(\sigma_2 + \tilde{\sigma}_2) = \sigma_1 \end{aligned}$$

然后同理可得

$$\sum_{j=5}^8 p_j = \tilde{\sigma}_1$$

于是就有

$$\sum_{j=1}^8 p_j = \sigma_1 + \tilde{\sigma}_1 = 1$$

对于一般的 n 而言，相应的计算是类似的。

那么既然样本属于每棵树各个叶节点的概率之和为 1，我们就能把样本 \mathbf{x} 属于某个叶节点 π_j 的概率 p_j 视为“对于样本 \mathbf{x} 而言 π_j 的权重”。如果此时每个 π_j 本身都“携带着”一个类别概率向量的话，那么我们就自然地将每棵树的输出 π 定义为这些类别概率向量的加权求和。

具体而言，假设现在的任务是 K 分类问题，那么每个叶节点 π_j 就都可以视为一个 K 维的概率向量：

$$\pi_j = (\pi_j^{(1)}, \pi_j^{(2)}, \dots, \pi_j^{(K)})^T$$

其中

$$\sum_{k=1}^K \pi_j^{(k)} = 1$$

且 $\pi_j^{(k)}$ 就代表着样本在叶节点 π_j 处属于第 k 类的概率。此时如果假设每棵树的总输出为

$$\pi = (\pi^{(1)}, \pi^{(2)}, \dots, \pi^{(K)})^T$$

那么所谓的加权求和就是

$$\pi = \sum_{j=1}^{2^{n+1}} p_j \pi_j = \left(\sum_{j=1}^{2^{n+1}} p_j \pi_j^{(1)}, \sum_{j=1}^{2^{n+1}} p_j \pi_j^{(2)}, \dots, \sum_{j=1}^{2^{n+1}} p_j \pi_j^{(K)} \right)^T$$

从而就有

$$\begin{aligned} \sum_{k=1}^K \pi^{(k)} &= \sum_{k=1}^K \sum_{j=1}^{2^{n+1}} p_j \pi_j^{(k)} = \sum_{j=1}^{2^{n+1}} \sum_{k=1}^K p_j \pi_j^{(k)} \\ &= \sum_{j=1}^{2^{n+1}} p_j \sum_{k=1}^K \pi_j^{(k)} = \sum_{j=1}^{2^{n+1}} p_j = 1 \end{aligned}$$

这意味着 DNDF 结构中每棵树的输出都会是一个概率向量，或说每棵树的输出都将会是一个概率输出。

为了方便叙述，这里我们额外定义一个类别概率矩阵 π^* ：

$$\boldsymbol{\pi}^* = \begin{bmatrix} \pi_1^{(1)} & \pi_1^{(2)} & \dots & \pi_1^{(K)} \\ \pi_2^{(1)} & \pi_2^{(2)} & \dots & \pi_2^{(K)} \\ \vdots & \vdots & \ddots & \vdots \\ \pi_{2^{n+1}}^{(1)} & \pi_{2^{n+1}}^{(2)} & \dots & \pi_{2^{n+1}}^{(K)} \end{bmatrix}$$

考虑到决策概率向量为

$$\boldsymbol{p} = (p_1, p_2, \dots, p_{2^{n+1}})^T$$

那么上述 $\boldsymbol{\pi}$ 的加权求和计算公式就能改写为

$$\boldsymbol{\pi} = \boldsymbol{p}^T \boldsymbol{\pi}^*$$

即每棵树的最终（概率）输出，其实就是对决策概率向量与类别概率矩阵进行矩阵乘法后得到的结果。

现在我们知道，DNDF 中每棵树的输出都是一个概率输出，所以 DNDF 结构本身的最终输出的定义方式就很自然、直观了：只需对这些树的概率输出取平均即可。具体而言，假设 DNDF 中一共有 M 棵树，并假设第 m 棵树对应的决策概率向量、类别概率矩阵分别为 $\boldsymbol{p}^{[m]}$ 、 $\boldsymbol{\pi}^{[m]*}$ ，那么第 m 棵树的概率输出即为

$$\boldsymbol{\pi}^{[m]} = \boldsymbol{p}^{[m]T} \boldsymbol{\pi}^{[m]*}$$

且 DNDF 的最终输出 $\boldsymbol{\pi}_{\text{DNDF}}$ 即可定义为

$$\boldsymbol{\pi}_{\text{DNDF}} = \frac{1}{M} \sum_{m=1}^M \boldsymbol{\pi}^{[m]}$$

由于每个 $\boldsymbol{\pi}^{[m]}$ 都是一个概率向量，所以不难看出 $\boldsymbol{\pi}_{\text{DNDF}}$ 也是一个概率向量，或说 DNDF 的最终输出即为概率输出。

5.3.2* DNDF 的具体实现

5.3.1 节我们大致介绍了 DNDF 的结构，这一节我们就来介绍它的具体实现。由于本节的内容可能会比较难，所以如果大家无法完全消化的话也没有关系，因为我们会在 5.3.3 节的开头对本节的关键内容进行直观的总结。如果能够理解这些直观总结的话，即使直接把本节跳过，也不会影响后续章节的阅读。

由前文的讨论不难得知，DNDF 的实现主要分为三个部分。

- 从特征层过渡到节点层（图 5.5 中的 FC 层）。
- 对节点层应用概率路由得到每棵树的决策概率向量。
- 定义类别概率矩阵并计算 DNDF 的最终（概率）输出。

同时，由于 DNDF 是相对独立的结构，所以我们应该将其抽象为一个单独的类。为此，我们先来看看它的初始化，然后再分别讨论上述三个核心部分的实现，如代码 5.2 所示。

代码 5.2 DNDF 的初始化: NNUtil.py

```

01 class DNDF:
02     """
03     初始化结构
04     n_class: 标签中的类别个数
05     如果是回归问题的话, 该值就应该被赋值为 1
06     n_tree: DNDF 中树的棵数, 默认为 16 棵
07     tree_depth: 每棵树的深度 (或说层数), 默认为 4
08     """
09     def __init__(self, n_class, n_tree=16, tree_depth=4):
10         self.n_class = n_class
11         self.n_tree, self.tree_depth = n_tree, tree_depth
12         # 由完全二叉树的性质可知, 当树的深度为 n 时,
13         # 叶节点的个数就是  $2^{n+1}$ , 中间节点数为叶节点数减 1
14         self.n_leaf = 2 ** (tree_depth + 1)
15         self.n_internals = self.n_leaf - 1

```

可以看到, 由于 DNDF 本身参数不多, 所以初始化的实现是非常简单的。

接下来就先看看第一个核心部分。虽然原论文所给出的特征层到节点层的过渡手段只是普通的线性映射, 但是如果考虑到后续的实现的话, 过渡的这一步需要做得更为细致。因此为了表达的连贯性, 我们就把该部分的实现稍微往后推迟一些。

然后来看看第二个核心部分, 此时我们需要实现出 DNDF 中的概率路由。由前文的讨论可知, 概率路由中用到的所有概率都用 σ 函数定义, 所以大体上可以把概率路由视为将节点层的激活函数设为 Sigmoid 函数后将各个神经元的非线性输出做了相应的整合。

具体而言, 假设特征层的输出是 $\mathbf{o}^{(m-1)}$, 那么第一个核心部分中对应的运算即为

$$\mathbf{f} = \mathbf{W}_0^{(m-1)} + \mathbf{W}^{(m-1)} \mathbf{o}^{(m-1)}$$

在这个公式中, \mathbf{f} 为节点层接收的输入。此时, 由于我们对节点层用了 Sigmoid 激活函数, 所以节点层的非线性输出即为

$$\mathbf{o}^{(m)} = \sigma(\mathbf{f}) = (\sigma(f_1), \sigma(f_2), \dots, \sigma(f_T))^T = (\sigma_1, \sigma_2, \dots, \sigma_T)^T$$

其中, $\mathbf{o}^{(m)}$ 的长度 T 是总的叶节点个数。不难得知, 假设我们的 DNDF 一共有 M 棵树、每棵树的深度为 n , 那么就有

$$T = M \times L$$

其中

$$L \triangleq 2^{n+1}$$

这是每棵树中叶节点的个数。同时, 考虑到

$$\tilde{\sigma}_i = 1 - \sigma_i$$

因此概率路由中所有用到的概率, 就都被包含在节点层的非线性输出 $\mathbf{o}^{(m)}$ 中了。我们下一步要做的, 就是把这些概率整合成概率路由的形式。考虑到概率路由的本质无非就是遍历所有

的决策路径并算出样本属于决策路径的概率 (p_j) 与其相应叶节点的类别概率向量 (π_j) 的乘积, 所以一种直观的做法就是: 采用第4章介绍过的先序遍历, 递归地计算出所有的概率。这种做法虽然肯定可行, 占用的内存也几乎是最少的, 但是它却没有利用到向量化运算, 所以真正运行起来会比较慢, 属于用时间复杂度 (https://en.wikipedia.org/wiki/Time_complexity) 换取空间复杂度 (https://en.wikipedia.org/wiki/Analysis_of_algorithms) 的做法。而如果想要加速 DNDF 概率路由的计算的话, 我们需要用一种比较巧妙的方式来完成 $\mathbf{o}^{(m)}$ 中各个概率的整合。仍以图 5.5 所示的 DNDF 结构中左边的树为例, 由前文的讨论可知:

$$\begin{aligned} p_1 &= \sigma_1 \sigma_2 \sigma_4, & p_2 &= \sigma_1 \sigma_2 \tilde{\sigma}_4, & p_3 &= \sigma_1 \tilde{\sigma}_2 \sigma_5, & p_4 &= \sigma_1 \tilde{\sigma}_2 \tilde{\sigma}_5 \\ p_5 &= \tilde{\sigma}_1 \sigma_3 \sigma_6, & p_6 &= \sigma_1 \sigma_3 \tilde{\sigma}_6, & p_7 &= \tilde{\sigma}_1 \tilde{\sigma}_3 \sigma_7, & p_8 &= \tilde{\sigma}_1 \tilde{\sigma}_3 \tilde{\sigma}_7 \end{aligned}$$

其中

$$\sigma_i = \sigma(d_i) = \sigma(f_i)$$

不难看出, σ_1 和 $\tilde{\sigma}_1$ (根节点对应的概率) 被重复利用了多次, σ_2 、 $\tilde{\sigma}_2$ 以及 σ_3 、 $\tilde{\sigma}_3$ (根节点的左右子节点对应的概率) 也被用了两次。对于一般的情况而言, 不难想象, 同样也是越靠近根节点的中间节点对应的概率 (后文统一简称为“中间概率”) 被利用的次数越多, 而越靠近叶节点的中间概率被利用的次数越少。

此外, 这些概率被重复利用的次数还有一个更有趣的性质: 它们都是 2 的幂次, 且某一层所有中间概率的利用次数都会比上一层所有中间概率的利用次数少 50%。具体而言, 假设现在树的深度为 n , 那么第 l 层的中间概率被重复利用的次数即为

$$n_l = 2^{n-l}$$

特别的, 由于根节点位于第 0 层, 所以它对应的概率被重复利用的次数为

$$n_1 = 2^n$$

这个特殊的性质能够让我们用一种更为巧妙的方式来整合出 DNDF 的概率路由。具体而言, 我们的目标是算出

$$\mathbf{p} = (p_1, p_2, \dots, p_8)^T$$

那么由前文的讨论就能发现

$$\mathbf{p} = \begin{matrix} \sigma_1 \\ \sigma_1 \\ \sigma_1 \\ \sigma_1 \\ \tilde{\sigma}_1 \\ \tilde{\sigma}_1 \\ \tilde{\sigma}_1 \\ \tilde{\sigma}_1 \end{matrix} \odot \begin{matrix} \sigma_2 \\ \sigma_2 \\ \tilde{\sigma}_2 \\ \tilde{\sigma}_2 \\ \sigma_3 \\ \sigma_3 \\ \tilde{\sigma}_3 \\ \tilde{\sigma}_3 \end{matrix} \odot \begin{matrix} \sigma_4 \\ \tilde{\sigma}_4 \\ \sigma_5 \\ \tilde{\sigma}_5 \\ \sigma_6 \\ \tilde{\sigma}_6 \\ \sigma_7 \\ \tilde{\sigma}_7 \end{matrix}$$

之所以它们能如此工整地对齐, 就是因为它们各自被重复利用的次数之间是 2 的幂次的关系。而且通过这种对齐的手段来计算 \mathbf{p} 的话, 不仅能使形式上更优雅, 而且效率也会比朴素的 (利用先序遍历的) 递归实现要快得多, 因为在 TensorFlow (乃至任何成熟的计算库) 中, 量化的算法都是被高度优化过的。

但是不难看出，这种利用重复性的向量化算法将会占用较多的内存，而且 TensorFlow 对这种“利用重复来进行优化”的做法并不完美支持，所以这又会额外地占用一些内存。总之，这是一种利用空间复杂度来换取时间复杂度的做法。因此具体是采用递归实现还是“重复实现”，还是要看具体的需求。不过在机器学习领域中，一般来说还是时间更为宝贵一些，所以我们接下来要展示的将会是重复实现。在此之前，我们先展示一下之前还没进行展示的、第一个核心部分的相应代码，为此我们需要实现出线性映射。

```
01 """
02     实现线性映射
03     net: 线性映射的输入
04     shape: 权值矩阵的 shape
05     appendix: 线性映射这一套运算步骤所属的 name scope 的后缀
06     pruner: 具体应用剪枝技术的实例
07 """
08 def fully_connected_linear(net, shape, appendix, pruner=None):
09     with tf.name_scope("Linear{}".format(appending)):
10         w_name = "W{}".format(appending)
11         w = init_w(shape, w_name)
12         # 如果使用剪枝技术的话，就调用 pruner 的相应方法进行剪枝
13         if pruner is not None:
14             w = pruner.prune w(*pruner.get w info(w))
15         b = init_b(shape[1], "b{}".format(appending))
16         return tf.add(tf.matmul(net, w), b, name="Linear{}".format(appending))
```

该函数是定义在 DNDF 类外面的公用函数，可以看到其实现与第 3 章 BasicNN 内部实现的线性映射方法很相似，只不过额外地引入了下一节将会介绍的剪枝技术。

接下来就是第一个核心部分的主体实现了，虽然不长，但却蕴含了比较复杂的逻辑。

```
01 """
02     实现特征层到节点层之间的过渡
03     net: 特征层的输出
04     dtype: 标识着 DNDF 真正的输出
05         其具体作用会在下一节给出，这里暂时按下不表
06     pruner: 具体应用剪枝技术的实例
07 """
08 def build_tree_projection(self, net, dtype, pruner):
09     with tf.name_scope("Tree Projection"):
10         # 定义一个存储节点层（铺平后的）非线性输出 $\mathbf{o}^{(m)}$ 的列表
11         flat_probabilities = []
12         # 记录特征层的神经元个数
13         fc_shape = net.shape[1].value
14         # 由于概率路由是针对每棵树来定义的，所以需要
15         # 对同一段操作循环 self.n_tree（树的棵数）次
16         for m in range(self.n_tree):
17             # 对于第 m 棵树而言
18             with tf.name_scope("Decisions"):
19                 # 利用 fully_connected_linear 和 tf.nn.sigmoid 函数
20                 # 来完成特征层到节点层的线性映射与非线性作用
```



```

21         p_left = tf.nn.sigmoid(fully_connected_linear(
22             net=net,
23             shape=[fc_shape, self.n_internals],
24             appendix=" tree mapping{} {}".format(m, dtype),
25             pruner=pruner
26         ))
27         # p_left 对应着样本在各个中间节点往左走的概率 $\sigma_1, \sigma_2, \dots$ 
28         # 所以要利用它来把 $\tilde{\sigma}_1 = 1 - \sigma_1, \tilde{\sigma}_2 = 1 - \sigma_2, \dots$ 这些
29         # 往右走的概率也算出来
30         p_right = 1 - p_left
31         # 最终的节点层就是这两组概率的合并
32         p_all = tf.concat([p_left, p_right], axis=1)
33         # 由于真正进行训练时, 接收的输入将会是一个 Mini Batch
34         # 所以我们要利用 tf.reshape 来把得到的结果铺平
35         flat_probabilities.append(tf.reshape(p_all, [-1]))
36     return flat_probabilities

```

为了更好地理解后文的实现, 接下来我们就先用一个具体的例子来说明上述代码的执行过程。假设现在有一个大小为 Q 的 Mini Batch, 它在经过了某个神经网络 (比如全连接神经网络或卷积神经网络) 后, 得到了特征层的输出:

$$\mathbf{O}^{(m-1)} = \begin{bmatrix} o_1^{(1)} & o_1^{(2)} & \dots & o_1^{(n^{(m-1)})} \\ o_2^{(1)} & o_2^{(2)} & \dots & o_2^{(n^{(m-1)})} \\ \vdots & \vdots & \ddots & \vdots \\ o_Q^{(1)} & o_Q^{(2)} & \dots & o_Q^{(n^{(m-1)})} \end{bmatrix}$$

不妨再假设 DNDf 中每棵树的深度为 n , 那么此时叶节点就有 $L \triangleq 2^{n+1}$ 个, 从而上述代码的循环体中, 第 21~26 行定义的 $\mathbf{p_left}$ 就是一个 $Q \times (L-1)$ 的矩阵, 该矩阵是特征层经过线性映射+激活函数后所产生的结果。为了简洁, 我们记 $L^* \triangleq L-1$, 从而 $\mathbf{p_left}$ 就是 $Q \times L^*$ 的矩阵, 且:

$$\mathbf{p_left} = \sigma(\mathbf{W}_0^{(m)} + \mathbf{O}^{(m-1)}\mathbf{W}^{(m)}) = \begin{bmatrix} \sigma_1^{(1)} & \sigma_1^{(2)} & \dots & \sigma_1^{(L^*)} \\ \sigma_2^{(1)} & \sigma_2^{(2)} & \dots & \sigma_2^{(L^*)} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_Q^{(1)} & \sigma_Q^{(2)} & \dots & \sigma_Q^{(L^*)} \end{bmatrix}$$

第 30 行定义的 $\mathbf{p_right}$ 则同样是一个 $Q \times L^*$ 的矩阵, 且

$$\mathbf{p_right} = 1 - \mathbf{p_left} = \begin{bmatrix} \tilde{\sigma}_1^{(1)} & \tilde{\sigma}_1^{(2)} & \dots & \tilde{\sigma}_1^{(L^*)} \\ \tilde{\sigma}_2^{(1)} & \tilde{\sigma}_2^{(2)} & \dots & \tilde{\sigma}_2^{(L^*)} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{\sigma}_Q^{(1)} & \tilde{\sigma}_Q^{(2)} & \dots & \tilde{\sigma}_Q^{(L^*)} \end{bmatrix}$$

有了 $\mathbf{p_left}$ 和 $\mathbf{p_right}$, 第 32 行定义的 $\mathbf{p_all}$ 就呼之欲出了:

$$\mathbf{p_all} = \begin{bmatrix} \sigma_1^{(1)} & \sigma_1^{(2)} & \dots & \sigma_1^{(L^*)} & \tilde{\sigma}_1^{(1)} & \tilde{\sigma}_1^{(2)} & \dots & \tilde{\sigma}_1^{(L^*)} \\ \sigma_2^{(1)} & \sigma_2^{(2)} & \dots & \sigma_2^{(L^*)} & \tilde{\sigma}_2^{(1)} & \tilde{\sigma}_2^{(2)} & \dots & \tilde{\sigma}_2^{(L^*)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma_Q^{(1)} & \sigma_Q^{(2)} & \dots & \sigma_Q^{(L^*)} & \tilde{\sigma}_Q^{(1)} & \tilde{\sigma}_Q^{(2)} & \dots & \tilde{\sigma}_Q^{(L^*)} \end{bmatrix}$$

然后我们在第 35 行处做了一个铺平的操作，从而在单次循环中，加入 `flat_probabilities` 的元素（不妨称之为 $\mathbf{p}^{(\text{flat})}$ ）都会是这样一个 Tensor：

$$\mathbf{p}^{(\text{flat})} = [\sigma_1^{(1)}, \dots, \sigma_1^{(L^*)}, \dots, \sigma_Q^{(1)}, \dots, \tilde{\sigma}_Q^{(L^*)}]$$

这个长度为 $2 \times L^* \times Q$ 的 Tensor，储存了 Mini Batch 在单棵树（比如说第 m 棵树）中进行概率路由的所有可能会用到的概率。

以上就是第一个核心部分的代码实现，以及它的返回值（`flat_probabilities`）所带来的信息的说明。接下来我们要做的，就是执行第二个核心工作：对节点层应用概率路由，并计算出决策概率向量。具体而言，我们需要按照之前“利用重复来进行加速”的方法，把第一个核心部分得到的 $\mathbf{p}^{(\text{flat})}$ 中的信息整合成这第 m 棵树的（概率）输出 $\boldsymbol{\pi}^{[m]}$ 。不失一般性，我们先把注意力集中在 Mini Batch 中的一个样本上，即先将 $\mathbf{p}^{(\text{flat})}$ 拆分为

$$\mathbf{p}^{(\text{flat})} \triangleq [\mathbf{p}_1^{(\text{flat})} \mathbf{p}_2^{(\text{flat})} \dots \mathbf{p}_Q^{(\text{flat})}]$$

其中

$$\mathbf{p}_q^{(\text{flat})} \triangleq [\sigma_q^{(1)}, \dots, \tilde{\sigma}_q^{(L^*)}], \quad q = 1, 2, \dots, Q$$

然后研究如何将 $\mathbf{p}_q^{(\text{flat})}$ 中的信息整合成第 q 个样本所对应的概率输出 $\boldsymbol{\pi}_q^{[m]}$ 。为了简洁，我们使用 $p_q^{(i)}$ 来代指 $\mathbf{p}_q^{(\text{flat})}$ 中的第 i 个概率，即

$$p_q^{(1)} = \sigma_q^{(1)}, \dots, p_q^{(L^*)} = \sigma_q^{(L^*)}; \quad p_q^{(L^*+1)} = \tilde{\sigma}_q^{(1)}, \dots, p_q^{(2L^*)} = \tilde{\sigma}_q^{(L^*)}$$

那么对于图 5.5 所示的结构而言，结合前文的讨论就能得出（注意此时 $L^* = 2^3 - 1 = 7$ ）

$$\mathbf{p}_q^{[m]} = \begin{matrix} p_q^{(1)} & p_q^{(2)} & p_q^{(4)} \\ p_q^{(1)} & p_q^{(2)} & p_q^{(L^*+4)} \\ p_q^{(1)} & p_q^{(L^*+2)} & p_q^{(5)} \\ p_q^{(1)} & p_q^{(L^*+2)} & p_q^{(L^*+5)} \\ p_q^{(L^*+1)} & p_q^{(3)} & p_q^{(6)} \\ p_q^{(L^*+1)} & p_q^{(3)} & p_q^{(L^*+6)} \\ p_q^{(L^*+1)} & p_q^{(L^*+3)} & p_q^{(7)} \\ p_q^{(L^*+1)} & p_q^{(L^*+3)} & p_q^{(L^*+7)} \end{matrix} \odot \quad \odot$$

其中， $\mathbf{p}_q^{[m]}$ 就是我们要计算的决策概率向量：它即为第 m 棵树中，第 k 个样本属于各个叶节点的概率所组成的概率向量。由于本例中一共有 $2^{n+1} = 8$ 个叶节点，所以可以看到 $\mathbf{p}_q^{[m]}$ 是一

个8维向量。

对于一般的情况来说， $\mathbf{p}_q^{[m]}$ 的计算方式也是类似的：

$$\mathbf{p}_q^{[m]} = \begin{matrix} p_q^{(1)} \\ p_q^{(1)} \\ \vdots \\ p_q^{(1)} \\ p_q^{(L^*+1)} \\ p_q^{(L^*+1)} \\ \vdots \\ p_q^{(L^*+1)} \end{matrix} \odot \begin{matrix} p_q^{(2)} \\ p_q^{(2)} \\ \vdots \\ p_q^{(L^*+2)} \\ p_q^{(3)} \\ p_q^{(3)} \\ \vdots \\ p_q^{(L^*+3)} \end{matrix} \odot \dots \odot \begin{matrix} p_q^{(2^n)} \\ p_q^{(L^*+2^n)} \\ \vdots \\ p_q^{(L^*+2^n+2^{n-1}-1)} \\ p_q^{(2^n+2^{n-1})} \\ p_q^{(L^*+2^n+2^{n-1})} \\ \vdots \\ p_q^{(L^*+2^{n+1}-1)} \end{matrix}$$

此时， $\mathbf{p}_q^{[m]}$ 就是一个 2^{n+1} 维的向量。如果从实现的角度来看的话，问题的关键在于如何定义出上式出现过的各个上标，从而利用 `tf.gather` 方法（https://www.tensorflow.org/api_docs/python/tf/gather），我们就能把相应位置的概率 $p_q^{(i)}$ 从 $\mathbf{p}_q^{(\text{flat})}$ 中提取出来了。此时，上述公式就应该写成 $\mathbf{p}_q^{(\text{flat})}$ 的数组下标的形式（注意在Python中是从0开始计数的，所以 $p_q^{(i)}$ 在 $\mathbf{p}_q^{(\text{flat})}$ 中的数组下标为 $i-1$ ）：

$$\mathbf{p}_q^{[m]} = \begin{matrix} 0 \\ 0 \\ \vdots \\ 0 \\ L^* \\ L^* \\ \vdots \\ L^* \end{matrix} \odot \begin{matrix} 1 \\ 1 \\ \vdots \\ L^*+1 \\ 2 \\ 2 \\ \vdots \\ L^*+2 \end{matrix} \odot \dots \odot \begin{matrix} 2^n-1 \\ L^*+2^n-1 \\ \vdots \\ L^*+2^n+2^{n-1}-2 \\ 2^n+2^{n-1}-1 \\ L^*+2^n+2^{n-1}-1 \\ \vdots \\ L^*+2^{n+1}-2 \end{matrix}$$

为了简洁，我们把上述公式中从左往右数的第 s 个列向量记为 $\mathbf{p}^{*(s-1)}$ ，则

$$\mathbf{p}_q^{[m]} = \mathbf{p}^{*(0)} \odot \mathbf{p}^{*(1)} \odot \dots \odot \mathbf{p}^{*(n)}$$

再考虑到 $\mathbf{p}_q^{(\text{flat})}$ 是最原始的概率 Tensor—— $\mathbf{p}^{(\text{flat})}$ 中的第 q 位，且每个 $\mathbf{p}_q^{(\text{flat})}$ 的长度都是 $2L^*$ ，那么令 $n_q \triangleq 2L^* \cdot (q-1)$ ，则上述公式写成 $\mathbf{p}^{(\text{flat})}$ 的数组下标的形式的话，就是

$$\mathbf{p}_q^{[m]} = (n_q + \mathbf{p}^{*(0)}) \odot (n_q + \mathbf{p}^{*(1)}) \odot \dots \odot (n_q + \mathbf{p}^{*(n)})$$

利用这个公式，我们就可以直接写出一个 Mini Batch 中的所有样本各自对应的决策概率向量所组成的“决策概率矩阵”了：

$$\mathbf{p}^{[m]} \triangleq \begin{bmatrix} (\mathbf{p}_1^{[m]})^T \\ (\mathbf{p}_2^{[m]})^T \\ \vdots \\ (\mathbf{p}_Q^{[m]})^T \end{bmatrix} = \begin{bmatrix} (n_1 + \mathbf{p}^{*(0)})^T \odot (n_1 + \mathbf{p}^{*(1)})^T \odot \dots \odot (n_1 + \mathbf{p}^{*(n)})^T \\ (n_2 + \mathbf{p}^{*(0)})^T \odot (n_2 + \mathbf{p}^{*(1)})^T \odot \dots \odot (n_2 + \mathbf{p}^{*(n)})^T \\ \vdots \\ (n_Q + \mathbf{p}^{*(0)})^T \odot (n_Q + \mathbf{p}^{*(1)})^T \odot \dots \odot (n_Q + \mathbf{p}^{*(n)})^T \end{bmatrix}$$

可以看到这是一个 $Q \times L$ 的矩阵，它的第 q 行就是第 q 个样本对应的长度为 L 的决策概率向量，而这个决策概率向量中的第 l 个元素就是第 q 个样本属于（第 m 棵树中）第 l 个叶节点的概率。

在实际的实现中，我们会采用 for 循环的方式来逐步地计算 $\mathbf{p}^{[m]}$ 。具体而言，我们会先把 $\mathbf{p}^{[m]}$ 初始化为

$$\mathbf{p}^{[m]} = \begin{bmatrix} (n_1 + \mathbf{p}^{*(0)})^T \\ (n_2 + \mathbf{p}^{*(0)})^T \\ \vdots \\ (n_Q + \mathbf{p}^{*(0)})^T \end{bmatrix}$$

然后开始循环（for $i = 1, 2, \dots, n$ do）

$$\mathbf{p}^{[m]} \odot = \begin{bmatrix} (n_1 + \mathbf{p}^{*(i)})^T \\ (n_2 + \mathbf{p}^{*(i)})^T \\ \vdots \\ (n_Q + \mathbf{p}^{*(i)})^T \end{bmatrix}$$

所以这里面其实只需要考虑如何获取

$$\begin{bmatrix} (n_1 + \mathbf{p}^{*(i)})^T \\ (n_2 + \mathbf{p}^{*(i)})^T \\ \vdots \\ (n_Q + \mathbf{p}^{*(i)})^T \end{bmatrix}, \quad i = 0, 1, 2, \dots, n$$

这 $n + 1$ 个矩阵即可。考虑到在这些矩阵中， n_1, n_2, \dots, n_Q 都是重复项，且 $\mathbf{p}^{*(i)}$ 是一个列向量，所以利用 numpy 的广播（Broadcasting）功能（<https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html>），我们就可以把这些矩阵统一写成

$$(n_1, n_2, \dots, n_Q)^T + \mathbf{p}^{*(i)T}, \quad i = 0, 1, 2, \dots, n$$

其中，我们已知

$$n_q = 2L^* \cdot (q - 1)$$

注意：在后文的代码中，我们会用 `batch_indices` 来代指 $(n_1, n_2, \dots, n_Q)^T$ 这个列向量。

所以接下来唯一还需要做的，就是计算出各个 $\mathbf{p}^{*(i)}$ 了。由 $\mathbf{p}^{*(i)}$ 的定义可知，它代指的是 DNDf 中（第 m 棵树的）第 i 层各个中间节点对应的、概率路由中需要用到的概率。比如对于根节点来说，它位于第 0 层，从而它对应的 $\mathbf{p}^{*(0)}$ 即为

$$\mathbf{p}^{*(0)} = \left(\overbrace{0, \dots, 0}^{2^n}, \overbrace{L^*, \dots, L^*}^{2^n} \right)^T$$

而对于根节点的左、右子节点而言，它们位于第 1 层，从而它们对应的 $\mathbf{p}^{*(1)}$ 即为

$$\mathbf{p}^{*(1)} = \left(\overbrace{1, \dots, 1}^{2^{n-1}}, \overbrace{L^* + 1, \dots, L^* + 1}^{2^{n-1}}, \overbrace{2, \dots, 2}^{2^{n-1}}, \overbrace{L^* + 2, \dots, L^* + 2}^{2^{n-1}} \right)^T$$

依此类推就可以得出，第 i 层对应的 $\mathbf{p}^{*(i)}$ 即为

$$\mathbf{p}^{*(i)} = \left(\overbrace{2^i - 1, \dots, 2^i - 1}^{2^{n-i}}, \overbrace{L^* + 2^i - 1, \dots, L^* + 2^i - 1}^{2^{n-i}}, \dots, \overbrace{L^* + 2^{i+1} - 2, \dots, L^* + 2^{i+1} - 2}^{2^{n-i}} \right)^T$$

如果利用 numpy 中的 np.repeat 函数 (<https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.repeat.html>)，上述 $\mathbf{p}^{*(i)}$ 就能表述为

$$\mathbf{p}^{*(i)} = \text{np.repeat}([2^i - 1, L^* + 2^i - 1, \dots, 2^{i+1} - 2, L^* + 2^{i+1} - 2], 2^{n-i})$$

同时，注意到

$$\tilde{\mathbf{p}}^{*(i)} \triangleq [2^i - 1, L^* + 2^i - 1, \dots, 2^{i+1} - 2, L^* + 2^{i+1} - 2]$$

可以拆分为

$$\tilde{\mathbf{p}}^{*(i)(1)} \triangleq [2^i - 1, 2^i - 1, \dots, 2^{i+1} - 2, 2^{i+1} - 2]$$

与

$$\tilde{\mathbf{p}}^{*(i)(2)} \triangleq [0, L^*, \dots, 0, L^*]$$

的和：

$$\tilde{\mathbf{p}}^{*(i)} = \tilde{\mathbf{p}}^{*(i)(1)} + \tilde{\mathbf{p}}^{*(i)(2)}$$

且对于 $\tilde{\mathbf{p}}^{*(i)(1)}$ 而言，也可以使用 np.repeat 与 np.arange (<https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.arange.html>) 来定义：

$$\tilde{\mathbf{p}}^{*(i)(1)} = \text{np.repeat}(\text{np.arange}(2^i - 1, 2^{i+1} - 2), 2)$$

而对于 $\tilde{\mathbf{p}}^{*(i)(2)}$ 而言，则可以用 np.tile (<https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.tile.html>) 来定义：

$$\tilde{\mathbf{p}}^{*(i)(2)} = \text{np.tile}([0, L^*], 2^i)$$

那么综上所述，相应的实现就不难理解了：

```

01  """
02      实现概率路由
03      flat_probabilities: 铺平后的、所有需要用到的概率
04      注：这是一个长度为  $M$  的 list，每个元素的长度为  $2 \times L^* \times Q$ 
05      其中， $M$  是 DNDF 中树的棵数， $Q$  是当前 Mini Batch 的大小
06      n_batch_placeholder: 代指当前 Mini Batch 大小  $Q$  的 placeholder
07      值得指出的是
08      1) self.n_leaf 为叶节点个数  $L$ 
09      2) self.n_internals 为中间节点个数  $L^*$ 
10      3) self.tree_depth 为每棵树的深度  $n$ 
11  """

```

```

12     def build_routes(self, flat_probabilities, n_batch_placeholder):
13         with tf.name_scope("Routes"):
14             # 利用 n_batch_placeholder, 定义出 $(n_1, n_2, \dots, n_Q)^T$ 
15             n_flat_prob = 2 * self.n_internals
16             batch_indices = tf.reshape(
17                 tf.range(0, n_flat_prob * n_batch_placeholder, n_flat_prob),
18                 [-1, 1]
19             )
20             # 用 n_repeat 代指第  $i$  层中 repeat 的次数 ( $2^{n-i}$ )
21             # 用 n_local_internals 代指第  $i$  层中中间节点的个数 ( $2^i$ )
22             # 由于根节点是第 0 层, 所以把它们分别初始化为
23             # ( $2^n = 2^{n+1} \div 2 = L / 2$ ) 和 ( $2^0 = 1$ )
24             n_repeat, n_local_internals = int(self.n_leaf * 0.5), 1
25             # 定义出 $\mathbf{p}^{(0)}$ 
26             increment_mask = np.repeat([0, self.n_internals], n_repeat)
27             # 初始化 ( $M$  棵树的) 决策概率矩阵
28             routes = [
29                 tf.gather(p_flat, batch_indices + increment_mask)
30                 for p_flat in flat_probabilities
31             ]
32             # 进入循环并计算
33             for depth in range(1, self.tree_depth + 1):
34                 # n_repeat 代指 $2^{n-i}$ , 所以每深一层它就要除以 2
35                 n_repeat = int(n_repeat * 0.5)
36                 # n_local_internals 代指 $2^i$ , 所以每深一层它就要乘以 2
37                 n_local_internals *= 2
38                 # 计算 $\tilde{\mathbf{p}}^{(i)(1)}$ 
39                 increment_mask = np.repeat(np.arange(
40                     n_local_internals - 1, 2 * n_local_internals - 1
41                 ), 2)
42                 # 叠加上 $\tilde{\mathbf{p}}^{(i)(2)}$ 
43                 increment_mask += np.tile(
44                     [0, self.n_internals], n_local_internals)
45                 # 再利用 np.repeat 算出 $\mathbf{p}^{(i)}$ 
46                 increment_mask = np.repeat(increment_mask, n_repeat)
47                 # 更新 ( $M$  棵树的) 决策概率矩阵
48                 for i, p_flat in enumerate(flat_probabilities):
49                     routes[i] *= tf.gather(
50                         p_flat, batch_indices + increment_mask)
51             # 返回 ( $M$  棵树的) 决策概率矩阵
52             return routes

```

以上就是第二个核心部分的实现, 也是整个 DNDF 中最难的部分。虽然代码量很少 (不算注释的话只有 24 行), 但其中蕴含的算法细节却非常多。

接下来看看最后一个核心部分的实现: 定义类别概率矩阵并计算 DNDF 的最终 (概率) 输出。对于这一部分, 原论文中的操作比较复杂: 它先将类别概率矩阵中的每一行 (即每棵树的类别概率向量) 初始化为均匀分布, 然后在训练过程中每训练完一定个数的 Mini Batch 之后,

用类似于不动点迭代的方式来更新类别概率矩阵。换句话说，原论文是割裂了类别概率矩阵和模型的其他参数的。在使用梯度下降训练模型的其他参数时，摠住类别概率矩阵不动；在使用不动点迭代更新类别概率矩阵时，摠住模型其他参数不动。

虽然这种做法从数学的角度来看比较漂亮，但是从实际应用上来看的话，可能就会存在三个比较麻烦的问题：

- 需要很小心地挑选训练速率等参数，否则类别概率矩阵经过不动点迭代后，可能会使得模型的其他参数训练变得很不稳定。
- 训练速度慢，因为不动点迭代需要做很多次前向传导算法。
- 不是完全的端到端训练。

为此，我们将会采用另一种做法：将叶节点对应的类别概率矩阵视为 TensorFlow 中的变量，并直接对其进行训练。具体而言，假设现在有 M 棵树，每棵树的深度都是 n ，且任务是 K 分类问题，那么第 m 棵树的 2^{n+1} 个叶节点对应的类别概率矩阵即为

$$\pi^{[m]*} = \begin{bmatrix} \pi_1^{(1)[m]} & \pi_1^{(2)[m]} & \dots & \pi_1^{(K)[m]} \\ \pi_2^{(1)[m]} & \pi_2^{(2)[m]} & \dots & \pi_2^{(K)[m]} \\ \vdots & \vdots & \ddots & \vdots \\ \pi_{2^{n+1}}^{(1)[m]} & \pi_{2^{n+1}}^{(1)[m]} & \dots & \pi_{2^{n+1}}^{(1)[m]} \end{bmatrix}$$

原论文中的做法是先将它初始化成一个均匀分布的概率矩阵

$$\pi^{[m]*} \leftarrow \begin{bmatrix} \frac{1}{K} & \frac{1}{K} & \dots & \frac{1}{K} \\ \frac{1}{K} & \frac{1}{K} & \dots & \frac{1}{K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{K} & \frac{1}{K} & \dots & \frac{1}{K} \end{bmatrix}$$

然后用类似于不动点迭代的方式来更新它

$$\pi^{[m]*} \leftarrow f(\pi^{[m]*})$$

其中 f 是具体的更新算法。而我们的做法则简单得多：只需要随机初始化一个 $2^{n+1} \times K$ 的矩阵 $\mathbf{W}^{[m]}$ ，继而把它设置为 TensorFlow 中的可训练变量（Variable）后，直接令

$$\pi^{[m]*} = \text{Softmax}(\mathbf{W}^{[m]})$$

即可。由于 TensorFlow 能够帮我们自动完成梯度下降的工作，所以此时随着训练过程的推进，TensorFlow 将会自动帮我们完成 $\pi^{[m]}$ 的训练工作，而无须像原论文中写的那样把 $\pi^{[m]}$ 的训练单独割裂出来。不难看出，这种做法其实很好地解决了上述三个问题。在下一节我们还会说明，这种做法背后的思想更是蕴含了非常多的可能性。

下面就来展示一下这种做法所对应的具体实现。由于这种做法说白了就只是定义一些矩阵而已，所以代码是非常简单的。

```

01     def build_leafs(self):
02         with tf.name_scope("Leafs"):
03             local_leafs = [
04                 tf.nn.softmax(w, name=" ClfLeafs{}".format(i))
05                 for i, w in enumerate([
06                     init_w([self.n_leaf, self.n_class], "RawClfLeafs")
07                     for _ in range(self.n_tree)
08                 ])
09             ]
09     return local_leafs

```

以上我们就完成了所有核心部分的代码实现，接下来要做的，是把它们进行整合，从而搭建出完整的 DNDF 结构。

```

01     """
02     搭建完整的 DNDF 结构
03     net: 特征层的输出
04     n_batch_placeholder: 代指当前 Mini Batch 大小 Q 的 placeholder
05     dtype: 标识着 DNDF 真正的输出
06     pruner: 具体应用剪枝技术的实例
07     """
08     def __call__(self, net, n_batch_placeholder,
09                 dtype="output", pruner=None):
10         name = "DNDF_{}".format(dtype)
11         with tf.variable_scope(name, reuse=tf.AUTO_REUSE):
12             # 获得铺平后的概率信息
13             flat_probabilities = self.build_tree_projection(net, dtype, pruner)
14             # 利用概率路由计算出 M 个决策概率矩阵
15             routes = self.build_routes(flat_probabilities, n_batch_placeholder)
16             # 将这 M 个决策概率矩阵合并为一个大矩阵
17             features = tf.concat(routes, 1, name="concat")
18             # 如果 dtype 取值为 feature，就直接返回这个大矩阵
19             if dtype == "feature":
20                 return features
21             # 否则，获取 M 棵树各自对应的类别概率矩阵
22             # 并把这 M 个类别概率矩阵合并为一个大矩阵
23             leafs = self.build_leafs()
24             leafs_matrix = tf.concat(leafs, 0, name="Prob_Concat")
25             # 然后利用各个决策概率矩阵和类别概率矩阵之间的矩阵乘法
26             # 来计算出 DNDF 最终的（概率）输出，注意由于要取均值
27             # 所以需要除以树的棵数 M
28             return tf.divide(
29                 tf.matmul(features, leafs_matrix),
30                 float(self.n_tree), name=name
31             )

```

5.3.3 DNDF 的应用场景

本节将会介绍一些 DNDF 的应用场景。我们不仅会对原论文中提到的应用场景进行介绍，还会介绍许多其他原论文中没有提及的应用场景。不过为了方便大家理解并真正掌握如何去应

用 DNDF，我们来对 DNDF 的一些重要概念做一个回顾。

首先是 DNDF 的定位。在原论文中，DNDF 被定位成“替代普通神经网络中最后的 Softmax 层”的结构。具体而言，对于一般的神经网络来说，其最后一层（即输出层）基本都会使用 Softmax 作为变换函数，从而得到一个概率输出：

$$\mathbf{o}^{(m)} = \phi^{(m)}(\mathbf{W}_0^{(m-1)} + \mathbf{W}^{(m-1)}\mathbf{o}^{(m-1)})$$

其中， $\phi^{(m)}$ 代指 Softmax 函数， $\mathbf{o}^{(m-1)}$ 则是倒数第二层（通常又被称为“特征层”）的输出。DNDF 的初衷就是把上述过程变为

$$\mathbf{o}^{(m)} = \text{DNDF}(\mathbf{o}^{(m-1)})$$

注意：这里的 m 代指第 m 层，而下述的 m 则代指第 m 棵树。

而 DNDF 的具体做法，就是通过 5.3.1 节中说过的决策概率向量

$$\mathbf{p}^{[m]} = (p_1^{[m]}, p_2^{[m]}, \dots, p_{2^{n+1}}^{[m]})^T$$

和类别概率矩阵

$$\boldsymbol{\pi}^{[m]*} = \begin{bmatrix} \pi_1^{(1)[m]} & \pi_1^{(2)[m]} & \dots & \pi_1^{(K)[m]} \\ \pi_2^{(1)[m]} & \pi_2^{(2)[m]} & \dots & \pi_2^{(K)[m]} \\ \vdots & \vdots & \ddots & \vdots \\ \pi_{2^{n+1}}^{(1)[m]} & \pi_{2^{n+1}}^{(1)[m]} & \dots & \pi_{2^{n+1}}^{(1)[m]} \end{bmatrix}$$

来计算出每棵树的最终（概率）输出：

$$\boldsymbol{\pi}^{[m]} = \boldsymbol{\pi}^{[m]T} \boldsymbol{\pi}^{[m]*}$$

并以此整合出 DNDF 的最终（概率）输出：

$$\boldsymbol{\pi}_{\text{DNDF}} = \frac{1}{M} \sum_{m=1}^M \boldsymbol{\pi}^{[m]}$$

所以这样来看的话，DNDF 的应用范围是极其广泛的：只要任务是分类问题的话，DNDF 就既能单独作为一个模型使用（此时特征层就是原始特征），又能作为任意一个模型的“增强器”（此时特征层对应的就是模型提取出来的特征）。这其实也正是原论文对 DNDF 的定位，因为所谓的“Softmax 层”其实也正是这样的东西，只不过它比起 DNDF 而言要弱一些。之所以 Softmax 层的应用如此广泛，主要原因之一就是没有既能做到适用面广，又能在方方面面都表现不错的替代品。如今 DNDF 的出现毫无疑问地撼动了 Softmax 层的地位，这也是笔者个人认为 DNDF 能斩获 2015 ICCV Best Paper 的一个重要因素。

不过，DNDF 是否只能用于分类问题呢？如果仔细思考其背后的原理，就会发现，DNDF 的应用场景其实远不止于此。具体而言，如果我们换一个角度来看 DNDF 模型的话，就能发现其本质为：

- 先是利用概率路由提取出了富有概率意义的特征（决策概率向量）。
- 然后使用富有概率意义的权值矩阵（类别概率矩阵）将这些特征线性映射成了输出。

换句话说，我们可以认为 DNDF 结构其实并不是一个模型，而只是一个特征提取器，只不过它可以在它自己提取出来的特征 ($\mathbf{p}^{[1]}, \mathbf{p}^{[2]}, \dots, \mathbf{p}^{[M]}$) 的基础上，用特殊的权值矩阵 ($\boldsymbol{\pi}^{[1]*}, \boldsymbol{\pi}^{[2]*}, \dots, \boldsymbol{\pi}^{[M]*}$) 线性映射来直接得到概率输出而已。如果我们把线性映射拿走，那么 DNDF 就完全可以被视为神经网络中的层结构——它能接收上一层的输入，并在内部进行一些非线性的处理后再输出一个新的特征。

具体而言，假设 DNDF 中一共有 M 棵树、每棵树的深度为 n 的话，那么 DNDF 输出的特征即为

$$\mathbf{p}^{(DNDF)} \triangleq [\mathbf{p}^{[1]} \mathbf{p}^{[2]} \dots \mathbf{p}^{[M]}] = (p_1^{[1]}, \dots, p_{2^{n+1}}^{[1]}, p_1^{[2]}, \dots, p_{2^{n+1}}^{[M]})^T$$

可以看出它是一个 $M \times 2^{n+1}$ 维的向量，且该向量拥有一些概率性质，因为它的 M 个组成部分

$$\mathbf{p}^{[1]}, \mathbf{p}^{[2]}, \dots, \mathbf{p}^{[M]}$$

本身都是概率向量：

$$\sum_{i=1}^{2^{n+1}} p_i^{[m]} = 1, \quad \forall m = 1, 2, \dots, M$$

总之，我们可以把 DNDF 视为神经网络中的层结构，它能够接收某个特征层的输出，并返回一个较为特殊的、有一定概率意义的特征向量。这就意味着 DNDF 至少多了如下两个应用场景：

- 当样本流过 DNDF 后，不一定非得像原论文所说的那样马上进入输出层，而可以进入另一个层结构。
- 不一定非得像原论文所说的那样做分类问题，而可以做回归问题。

对于这两个额外的应用场景，相应的实现都是比较简单的。其中，我们之前频繁用到的却一直没有详细说明了、标识着 DNDF 真正的输出的参数 `dtype`，就是直接用来实现第一个额外的应用场景的。事实上，在 5.3.2 节最后一段代码的注释中，我们也已经给出过相应的说明：当 `dtype` 取值为 `feature` 的时候，就直接返回 M 个决策概率矩阵合并而成的大矩阵，因为这个大矩阵其实就是 $\mathbf{p}^{(DNDF)}$ ；而当 `dtype` 不取值为 `feature` 时，才按照原论文中的说法，返回一个概率输出。

在此基础上，第二个额外的应用场景——将 DNDF 应用在回归问题上——就是一个普通的拓展了。因为既然 DNDF 能被视为层结构的话，那么普通的神经网络该怎么做回归问题，DNDF 就能怎么做回归问题。唯一需要注意的是，由于 DNDF 输出的特征富有概率意义，所以 DNDF 在做分类时采用的权值矩阵也是富有概率意义的类别概率矩阵；而在回归问题中，由于样本中的标签不再是一个 `OneHot` 向量，而是一个连续的、取值范围未知的标量，所以我们就需要把带有概率意义的权值矩阵换回普通的权值矩阵。

具体而言，假设 DNDF 输出的特征为

$$\mathbf{p}^{(DNDF)} = (p_1^{[1]}, \dots, p_{2^{n+1}}^{[1]}, p_1^{[2]}, \dots, p_{2^{n+1}}^{[M]})^T$$

那么在做回归问题时，DNDF 采用的权值矩阵就应该是一个 $1 \times (M \cdot 2^{n+1})$ 的矩阵：

$$\mathbf{W} = [w^{(1)}, w^{(2)}, \dots, w^{(M \cdot 2^{n+1})}]$$

从而 DNDF 的最终输出就是

$$o^{(m)} = \frac{1}{M} \cdot \mathbf{W} \mathbf{p}^{(\text{DNDF})}$$

有意思的是，如果我们进一步对 $\mathbf{W}^{(\text{Reg})}$ 中的各个元素进行分组

$$\mathbf{W} = [w_1^{[1]}, \dots, w_{2^{n+1}}^{[1]}, w_1^{[2]}, \dots, w_{2^{n+1}}^{[M]}] \triangleq [\mathbf{w}^{[1]} \mathbf{w}^{[2]} \dots \mathbf{w}^{[M]}]$$

那么就能把最终输出写成

$$o^{(m)} = \frac{1}{M} \sum_{m=1}^M \mathbf{w}^{[m]} p^{[m]}$$

换句话说，我们可以认为 DNDF 在做回归时的输出是一些“加权概率分布”的均值，这就意味着 DNDF 甚至能让回归问题也获得一定的概率意义。

以上就是理论层面的介绍。至于具体的实现的话，由上述的讨论不难看出，我们只需对第三个核心部分，即定义类别概率向量所对应的实现做出一些小改动即可。

```

01     def build_leafs(self):
02         with tf.name_scope("Leafs"):
03             # 如果是回归问题的话，就定义一些普通的权值矩阵
04             if self.n_class == 1:
05                 local_leafs = [
06                     init_w([self.n_leaf, 1], "RegLeaf{}".format(i))
07                     for i in range(self.n_tree)]
08             # 否则，就按照之前的方法定义出
09             # 富有概率意义的权值矩阵（类别概率矩阵）
10             else:
11                 local_leafs = [
12                     tf.nn.softmax(w, name="ClfLeafs{}".format(i))
13                     for i, w in enumerate([
14                         init_w([self.n_leaf, self.n_class], "RawClfLeafs")
15                         for _ in range(self.n_tree)
16                     ])]
17             return local_leafs

```

5.3.4* DNDF 的结构内涵

前三节大致叙述了 DNDF 的结构、实现与应用场景。虽然从中确实能了解 DNDF 的具体原理，但对于为什么 DNDF 能有好的效果可能就无法获得直观的感受了。所以这一节我们会着重将它与决策树及其集成算法进行比较，希望这种比较能让大家更好地理解 DNDF 结构（在笔者眼中）的真正内涵。本节的内容同样具有一定难度，可能需要具有一定的机器学习基础才能完全吃透；不过我们会在本节的最后给出要点总结，所以即使没有相应的基础也没关系，大家只需对这些总结有一个直观理解即可。

在 5.3.1 节的开头我们曾经说过, DNDF 能非常自然地在神经网络中引入树的结构。这一点虽然非常显然(比如图 5.5 直接就把树画了出来),但需要指出的是, DNDF 引入树的结构的方式不仅流于表面,而是把决策树的性质也引入了进来。特别是当决策树是二叉树(比如说 scikit-learn 生成的决策树)时, DNDF 就能从理论上包容它的表达能力。

具体而言,假设某个二叉决策树 T 的深度为 $n^{(\text{DT})}$ 、中间节点数为 N ,且这 N 个中间节点对应的超平面为

$$x^{(i_j)} - \tilde{\epsilon}_j = 0, \quad j = 1, 2, \dots, N$$

其中, i_j 和 $\tilde{\epsilon}_j$ 分别是第 i 个中间节点所挑选的特征维度和阈值。那么此时在 DNDF 中,我们只需要用 1 棵深度为

$$n = n^{(\text{DT})}$$

的树 $T^{(\text{DNDF})}$, 就能包容 T 的表达能力。具体而言,假设 $T^{(\text{DNDF})}$ 的中间节点为

$$d_1, d_2, \dots, d_{2^{n+1}}$$

由于 $T^{(\text{DNDF})}$ 与二叉决策树 T 的深度一致,所以就能在 $T^{(\text{DNDF})}$ 中选出

$$d_1, d_2, \dots, d_N$$

这 N 个中间节点,使得它们组成的子树在拓扑结构上和 T 同构。又考虑到

$$d_j = \sum_{i=1}^n w_j^{(i)} x^{(i)} + w_j^{(0)}, \quad j = 1, 2, \dots, N$$

其中, $w_j^{(i)}$ 和 $w_j^{(0)}$ 分别是 DNDF 中特征层过渡到节点层时所用的权值和偏置量。那么此时只需令

$$w_j^{(i)} = \begin{cases} -C & , \quad \text{if } i = i_j \\ 0 & , \quad \text{if } i \neq i_j \end{cases}, \quad w_j^{(0)} = C\tilde{\epsilon}_j$$

即可,其中 C 是一个大于 0 的常数。此时就有

$$d_j = -Cx^{(i_j)} + C\tilde{\epsilon}_j = -C(x^{(i_j)} - \tilde{\epsilon}_j)$$

注意, DNDF 在节点层处用的激活函数是 Sigmoid,并视激活函数值为样本往左走的概率,从而样本在中间节点 d_j 处往左走的概率即为

$$p^{\text{DNDF}}(x \in d_j^{\text{left}}) = \sigma(d_j) = \sigma(-C(x^{(i_j)} - \tilde{\epsilon}_j))$$

由 σ 函数的性质不难看出,当 C 是一个非常大的常数时,就有

$$p^{\text{DNDF}}(x \in d_j^{\text{left}}) \rightarrow \begin{cases} 1 & , \quad \text{if } x^{(i_j)} < \tilde{\epsilon}_j \\ 0 & , \quad \text{if } x^{(i_j)} > \tilde{\epsilon}_j \end{cases}$$

对于决策树而言,会有

$$p^{\text{DT}}(x \in d_j^{\text{left}}) = \begin{cases} 1 & , \quad \text{if } x^{(i_j)} < \tilde{\epsilon}_j \\ 0 & , \quad \text{if } x^{(i_j)} \geq \tilde{\epsilon}_j \end{cases}$$

于是就可以看出，只要 C 取得足够大，DNDF 就能以任意的精度逼近决策树的表达能力，即 DNDF 从理论上就能包容决策树模型。

注意：在 DNDF 中，由于 Sigmoid 在 0 点处的函数值总为 0.5，所以当 $x^{(i_j)} = \xi_j$ 时，就总有 $p^{\text{DNDF}}(x \in d_j^{\text{left}}) = 0.5$ 。不过由于当 $x^{(i_j)}$ 是连续型特征时，它取到单个值的概率为 0，所以这种情况在统计意义上可以忽略不计。

此外，我们曾在 5.3.1 节中说过 DNDF 对应的决策面可以是任意的超平面，这一点的证明其实与证明 DNDF 能包容决策树模型的过程是一致的。具体而言，假设现在有一个超平面的表达式为

$$\Pi : a_0 + a_1x^{(1)} + \cdots + a_nx^{(n)}$$

那么如果能让 DNDF 的中间节点 d_j 对应的决策面为 Π 的话，就只需令

$$w_j^{(i)} = -Ca_i, \quad i = 0, 1, \dots, n$$

即可。此时当 C 足够大时，就有

$$p^{\text{DNDF}}(x \in d_j^{\text{left}}) \rightarrow \begin{cases} 1 & , \text{ if } a_0 + a_1x^{(1)} + \cdots + a_nx^{(n)} < 0 \\ 0 & , \text{ if } a_0 + a_1x^{(1)} + \cdots + a_nx^{(n)} > 0 \end{cases}$$

即此时 d_j 就表达出了超平面 Π 。总之，DNDF 不仅能够从理论上拥有决策树的表达能力，还能破除决策树中的硬边界问题（因为它采取了概率路由）和决策面垂直于坐标轴的问题（因为决策面可以是任意超平面），甚至能和神经网络无缝对接（因为它能作为神经网络中的层结构出现），所以至少从理论上来说，DNDF 要比第 4 章介绍的决策树转换而得的神经网络 NN_{DT} 要更优雅、更强大。

以上就是 DNDF 全部的、较为详尽的介绍了。作为补充，笔者在这里额外提供一个 DNDF 可能可以继续研究的方向：对 DNDF 应用集成算法。该想法其实是非常自然的，毕竟现在流行的、效果突出的集成算法基本都是基于树的集成算法，它们会依次训练许多棵树，然后以一定的规则将它们整合成一个最终的模型。那么既然 DNDF 中的一棵树就能包容任意的决策树，且 DNDF 本身就含有 M 棵树的话，尝试使用集成算法来初始化这 M 棵树就是一个水到渠成的思路。

不过这毕竟只是一个很宽泛的方向，真的要进行研究的话需要补充很多具体的细节，比如怎样去训练单棵树、训练单棵树时是否需要固定住特征层、训练完之后怎样进行集成……我们会在 5.5 节介绍各种技术的互补性时，提供一种简单的、具体可行的、对 DNDF 应用集成算法的做法，这里暂时按下不表。

在本节的最后，我们对本节的内容做一个小结以方便大家记忆、查阅：

- DNDF 中的一棵树就能包容决策树模型。
- DNDF 中，每棵树的中间节点都可以表达任意的超平面。
- 可以考虑对 DNDF 中的 M 棵树应用集成算法，因为这种做法可能能够增加 DNDF 中各棵树的多样性。

5.4 神经网络中的剪枝

5.2 节介绍的 Wide and Deep 改变了普通神经网络的结构，5.3 节介绍的 DNDF 增强了普通神经网络的概率意义，但是它们都没有改变普通神经网络中的一个痛点：层与层之间那全连接的连接方式。而事实上，近些年获得了重大成功的 CNN 和 RNN 都对 DNN 这种简单粗暴的全连接做出了改进，其中 CNN 用了局部连接、RNN 用了权值共享。

那么对于普通的神经网络而言，如果不改变其结构本质的前提下，应该如何改进全连接这种连接方式呢？一个比较自然的想法就是希望用“局部连接”去代替它：与全连接不同，每一层的神经元都不会受上一层的所有神经元影响，而只会受上一层的某些神经元影响。这意味着我们希望神经网络能够拥有这样一个性质：随着训练的进行，神经网络能够检查出全连接中相对“冗余”的连接，并挖掘出真正重要的连接。

之所以我们希望神经网络的连接方式是局部连接而不是全连接，主要是出于这样的考虑：如果把一个神经元看作一个特征整合单元的话，那么当连接方式是全连接时，所有特征整合单元看到的原始特征都是一致的，这就导致这些特征整合单元整合出来的特征可能会趋于“同质”，从而造成大量冗余。而如果连接方式是局部连接的话，每个特征整合单元看到的原始特征将会各不相同，从而它们整合出来的特征也会各不相同，这就减少了该层输出特征的冗余并增加了多样性。

而把神经网络的连接方式从全连接变成局部连接的方法，就要用到之前提到过不少次的剪枝技术了。神经网络中的剪枝虽然和第 2 章说过的决策树的剪枝不同，但它们的思想是类似的：通过剪掉模型中多余的部分，让模型的复杂度降低，从而提高模型的泛化能力。在决策树中，剪枝剪掉的是某棵子树；而在神经网络中，剪枝剪掉的就是神经元之间的连接。

5.4.1 Surgery 算法概述

我们将采用的剪枝算法是受了 *Dynamic Network Surgery for Efficient DNNs* (<https://arxiv.org/pdf/1608.04493.pdf>) 这篇论文（后文统一简称其为“Surgery”）的启发而设计的算法。Surgery 这篇论文的目标是压缩模型，它发明了一种动态剪枝的手法，使得神经网络能够在维持性能不下降的前提下将全连接变成局部连接。此时权值矩阵就变成了稀疏矩阵，从而我们就能用更少的空间来储存它。

不过，虽然它确实能够将全连接变成局部连接，但是其（压缩模型的）初衷却和我们（提高神经网络的性能）的初衷是有些冲突的。也正因此，Surgery 中提出的剪枝方法其实基本无法提升神经网络的性能，而只能够节省一些存储空间。因此如果想要达成我们的初衷，需要对 Surgery 提出的剪枝做出改进。

下面就先来介绍 Surgery 提出的剪枝方案。通俗来说，它会根据每个权值和整个权值矩阵的关系来决定是否应该把相应的权值“剪掉”。同时，即使某个权值被“剪掉”了，也有可能在今后的训练过程中恢复回来。具体而言，假设将要应用剪枝技术的权值矩阵为

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1q} \\ w_{21} & w_{22} & \dots & w_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p1} & w_{p2} & \dots & w_{pq} \end{bmatrix}$$

那么 Surgery 的做法，就是维护一个二值矩阵 T :

$$\mathbf{T} = \begin{bmatrix} t_{11} & t_{12} & \dots & t_{1q} \\ t_{21} & t_{22} & \dots & t_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ t_{p1} & t_{p2} & \dots & t_{pq} \end{bmatrix}$$

其中

$$t_{ij} \in \{0,1\}, \quad i = 1 \sim p, j = 1 \sim q$$

注意：我们通常会称和某个矩阵形状一致的二值矩阵为该矩阵的 mask，比如说上述的 T 就是 W 的 mask。

然后，神经网络实际应用的权值矩阵将会是 \mathbf{W} 和 \mathbf{T} 的 Hadamard 乘积：

$$\widetilde{\mathbf{W}} \triangleq \mathbf{W} \odot \mathbf{T}$$

不难看出，如果想把 \mathbf{W} 中的 w_{ij} 剪掉的话，只需把 t_{ij} 从 1 变成 0 即可；而如果后期又想恢复它的话，则只需把 t_{ij} 从 0 变回 1。所以通过动态地维护 \mathbf{T} ，我们就能完成权值矩阵 \mathbf{W} 的动态剪枝。不过需要指出的是，这个 T 不是一个固定的矩阵或是一个可以优化的参数，而是 \mathbf{W} 的一个函数

$$\mathbf{T} = h(\mathbf{W})$$

虽然 Surgery 这篇论文并没有显式给出 h 的定义，不过笔者通过阅读论文作者开源出来的代码（C++，caffe），发现 Surgery 定义的 h 拥有如下的形式：

$$h(w_{ij}) = \begin{cases} 0 & , \quad \text{if } (T_{ij} = 1) \wedge (|w_{ij}| \leq 0.9 \cdot (\text{mu} + \beta \text{std})) \\ 1 & , \quad \text{if } (T_{ij} = 0) \wedge (|w_{ij}| \geq 1.1 \cdot (\text{mu} + \beta \text{std})) \\ T_{ij} & , \quad \text{else} \end{cases}$$

其中，0.9、1.1和 β 都是预先设置好的超参数，mu 和 std 则分别是 \mathbf{W} 中各个元素的绝对值均值和绝对值标准差，即

$$\text{mu} = \sum_{i=1}^p \sum_{j=1}^q |w_{ij}|$$

$$\text{std} = \sqrt{\frac{1}{pq-1} \cdot \sum_{i=1}^p \sum_{j=1}^q (|w_{ij}| - \text{mu})^2}$$

同时，Surgery 采用了依概率更新的方式来更新 T 。具体而言，它在每次迭代时会有 p 的概率去更新 T ，其中 p 是当前迭代步数的函数：

$$p = f(\text{iter})$$

Surgery 给出了函数 f 的限制：它需要是单调不减的函数，而且需要满足 $f(0) = 1$ 。不过 Surgery 没有给出 f 的具体形式，所以 f 的选取可能并不是一个特别重要的事情。

5.4.2 Surgery 算法改进

5.4.1 节简介的 Surgery 算法拥有形式简洁、剪枝效果显著的优点（原论文给出的实验结果表明它甚至能把 99% 的权值都剪掉）。但是，Surgery 算法毕竟旨在压缩模型而非提升模型性能，所以它剪枝的方式是比较“硬”的：对于每个权值而言，一旦它被剪掉就是直接变成 0，没有过渡的过程。这种剪枝手段虽然能够在特征较差、噪声较大的数据集上表现优异（因为它能“强硬”地把所有噪声特征对应的权值都剪掉），但是也经常在特征已经做得较好的数据集上出现性能损失（因为它会把好的特征对应的权值也完全剪掉）。

为此，笔者在 Surgery 的基础上，开发出了“软剪枝”技术。软剪枝的主要思想其实非常自然：希望能够引入权值衰减过程，从而降低模型的突变性。实验证明，软剪枝不仅在大多数较差的数据集上能达到 Surgery 这种“硬剪枝”的效果，而且在较好的数据集上也几乎没有性能损失。

注意：我们会在 5.6 节给出人造数据集上的实验结果，并在第 7 章给出许多真实数据集上的实验结果。

所谓的软剪枝，其实只是把函数 h 从 Surgery 所定义的

$$h(W_{ij}) = \begin{cases} 0 & , \quad \text{if } (T_{ij} = 1) \wedge (|W_{ij}| \leq 0.9 \cdot (\text{mu} + \beta \text{std})) \\ 1 & , \quad \text{if } (T_{ij} = 0) \wedge (|W_{ij}| \geq 1.1 \cdot (\text{mu} + \beta \text{std})) \\ T_{ij} & , \quad \text{else} \end{cases}$$

变成看上去有些复杂的

$$\tilde{T}_{ij} \triangleq \min \left(r, \beta \log \left(\max \left(\epsilon, \frac{|W_{ij}|}{\gamma \cdot \text{mu}} \right) \right) \right)$$

$$h(W_{ij}) = T_{ij} = \max \left(\frac{\alpha}{\beta} \cdot \tilde{T}_{ij}, \tilde{T}_{ij} \right)$$

而已，其中 α 、 β 、 γ 和 r 都是预先设置好的超参数， ϵ 是保证数值稳定性的小值， mu 则仍然代指 W 中各个元素的绝对值均值。虽然这个公式看上去很复杂，但从图像来看的话是非常直观的，如图 5.6、图 5.7 所示，横、纵坐标分别是原始权值 W_{ij} 的大小和剪枝后的权值 $\tilde{W}_{ij} = W_{ij} \cdot T_{ij}$ 的大小。

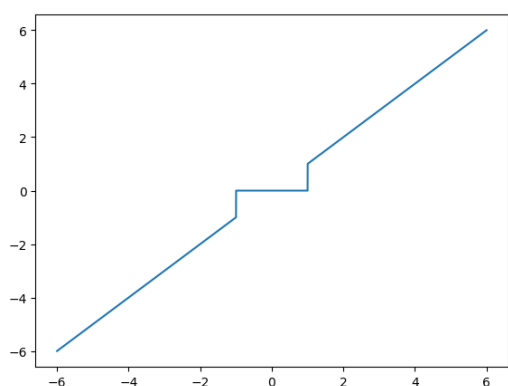


图5.6 Surgery的剪枝图像

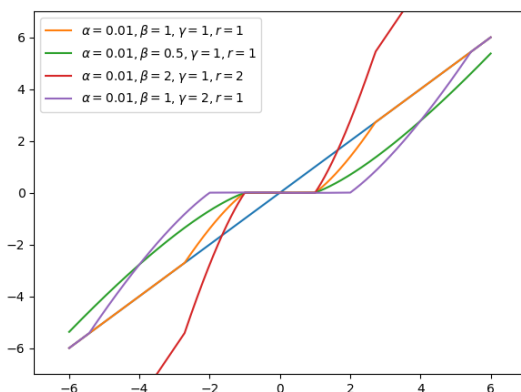


图5.7 软剪枝在各种参数设置下的剪枝图像

可以看到，Surgery 会有一个“突然变 0”的地方，而软剪枝在变 0 之前则会有一个缓慢的衰减过程。同时，软剪枝裁剪权值时其实是利用了

$$\lim_{x \rightarrow 0} x \log x = 0$$

这个性质（我们会在后文补充说明为什么软剪枝利用了这个性质），因此虽然当权值本身是 0 时，经过软剪枝后它仍然会取 0，但是当权值取在 0 点附近时，它被软剪枝裁剪后就会变成一个很小的数（比如， 10^{-3} 量级的数）而不是变成 0 本身。相反，被 Surgery 剪掉的权值将会直接变成 0。这就意味着 Surgery 在模型压缩上毫无疑问是远胜软剪枝的，但也正因为 Surgery 想要做到这种真正的模型压缩，所以它的剪枝过程非常突兀。当权值矩阵中的权值整体较大时，Surgery 可能就会直接将一个较大的权值剪成 0，从而对模型的性能造成较大冲击。

此外，软剪枝还有一个优点——它在一定程度上是包容 Surgery 的。事实上从图 5.7 可以看出，软剪枝中的各个参数拥有很好的直观意义。

- α ：调整权值被软剪枝裁剪后的大小， α 越大则裁剪后的权值越大（这一点从图中无法看出来，但是从公式 $T_{ij} = \max\left(\frac{\alpha}{\beta} \cdot \tilde{T}_{ij}, \tilde{T}_{ij}\right)$ 中可以比较直观地认识到）。
- β ：权值在被软剪枝裁剪前衰减的速度， β 越大则权值衰减得越快。
- γ ：权值被软剪枝裁剪的阈值相比于 μ 而言的倍数。换句话说，权值会在 $\gamma \cdot \mu$ 处被软剪枝裁剪。
- r ：权值在没有被剪枝时能够超出原始权值的倍数。换句话说，权值经过软剪枝后不可能超过原来的 r 倍。

那么从直观上来说，我们只需令

$$\alpha \rightarrow 0, \quad \beta \rightarrow \infty, \quad \gamma = r = 1$$

意味着软剪枝能够逼近 Surgery 的表现，因为

- 权值在被软剪枝裁剪后趋于 0。
- 权值衰减的速度趋于无穷大。
- 权值会在 μ 处被裁剪，且在其余情况下不可能超过原来的取值。

而从（不完全严谨的）理论的角度来看的话也确实如此，因为

$$\begin{aligned}\tilde{T}_{ij} &\triangleq \min\left(r, \beta \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\gamma \cdot \mu}\right)\right)\right) \\ &= \min\left(1, \beta \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right)\right)\end{aligned}$$

且

$$\log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) \begin{cases} < 0 & , \text{ if } |W_{ij}| < \mu \\ = 0 & , \text{ if } |W_{ij}| = \mu \\ > 0 & , \text{ if } |W_{ij}| > \mu \end{cases}$$

那么由于 $\beta \rightarrow \infty$ ，所以必有 $\alpha < \beta$ ，且

$$\beta \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) \begin{cases} \rightarrow -\infty & , \text{ if } |W_{ij}| < \mu \\ = 0 & , \text{ if } |W_{ij}| = \mu \\ \rightarrow \infty & , \text{ if } |W_{ij}| > \mu \end{cases}$$

从而

$$\min\left(1, \beta \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right)\right) = \begin{cases} \beta \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) & , \text{ if } |W_{ij}| \leq \mu \\ 1 & , \text{ if } |W_{ij}| > \mu \end{cases}$$

即

$$\tilde{T}_{ij} = \begin{cases} \beta \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) & , \text{ if } |W_{ij}| \leq \mu \\ 1 & , \text{ if } |W_{ij}| > \mu \end{cases}$$

从而

$$\frac{\alpha}{\beta} \cdot \tilde{T}_{ij} = \begin{cases} \alpha \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) & , \text{ if } |W_{ij}| \geq \mu \\ \frac{\alpha}{\beta} & , \text{ if } |W_{ij}| > \mu \end{cases}$$

又由于 $\alpha < \beta$ ，以及当 $|W_{ij}| \leq \mu$ 时 $\log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) \leq 0$ ，从而就有

$$\max\left(\frac{\alpha}{\beta} \cdot \tilde{T}_{ij}, \tilde{T}_{ij}\right) = \begin{cases} \alpha \log\left(\max\left(\epsilon, \frac{|W_{ij}|}{\mu}\right)\right) & , \text{ if } |W_{ij}| \leq \mu \\ 1 & , \text{ if } |W_{ij}| > \mu \end{cases}$$

即

$$T_{ij} = \begin{cases} \alpha \log \left(\max \left(\epsilon, \frac{|W_{ij}|}{\mu} \right) \right) & , \text{ if } |W_{ij}| \leq \mu \\ 1 & , \text{ if } |W_{ij}| > \mu \end{cases}$$

于是就有

$$\tilde{W}_{ij} = W_{ij} \cdot T_{ij} = \begin{cases} \alpha W_{ij} \log \left(\max \left(\epsilon, \frac{|W_{ij}|}{\mu} \right) \right) & , \text{ if } |W_{ij}| \leq \mu \\ W_{ij} & , \text{ if } |W_{ij}| > \mu \end{cases}$$

从而由

$$\lim_{x \rightarrow 0} x \log x = 0$$

可知，无论 α 取值为多少，都会有

$$\lim_{W_{ij} \rightarrow 0} \tilde{W}_{ij} = 0$$

即当 W_{ij} 取 0 时，被软剪枝裁剪后的权值 \tilde{W}_{ij} 也必定取 0。但是，如果要在此基础上限定 \tilde{W}_{ij} 在 0 和 μ 之间也都取 0，或说离 0 任意接近的话，就需要令 $\alpha \rightarrow 0$ 了。

最后需要特别指出的是，无论是原始的 Surgery 还是改进后的软剪枝，它们都秉承着这样一个理念：“权值的绝对值越大就意味着该权值越重要”（即权值的绝对值大小与重要性呈正相关）。虽然该理念非常自然，但它并不总是成立的。我们会在 5.4.4 节的最后给出某种特殊情况下的证明，这里暂时按下不表。

5.4.3 软剪枝的具体实现

由上一节的讨论可知，软剪枝的本质可以视为对权值矩阵 W 做了一个变换：

$$\begin{aligned} \tilde{T}_{ij} &\leftarrow \min \left(r, \beta \log \left(\max \left(\epsilon, \frac{|W_{ij}|}{\gamma \cdot \mu} \right) \right) \right) \\ W_{ij} &\leftarrow W_{ij} \cdot \max \left(\frac{\alpha}{\beta} \cdot \tilde{T}_{ij}, \tilde{T}_{ij} \right) \end{aligned}$$

所以在具体实现上，我们只需要把这个变换中的公式写进 TensorFlow 的计算图中即可。为了简洁，下面只展现软剪枝的核心代码部分（参见代码 5.3），对完整的代码感兴趣的读者则可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/NNUtil.py。

注意：如果想要使用 Surgery 的话，需要完整版本的 Pruner。

代码 5.3 软剪枝的核心实现：NNUtil.py

```
01 class Pruner:
02     """
03     初始化结构
```

```

04     alpha, beta, gamma, r, eps:  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $r$ ,  $\epsilon$ 
05     其中,  $r$  默认为 1、 $\epsilon$  默认为  $1e-12$ 
06     """
07     def init (self, alpha=None, beta=None, gamma=None,
08               r=1., eps=1e-12):
09         self.alpha, self.beta, self.gamma, self.r, self.eps = alpha, beta, gamma, r, eps
10         # 定义一个存储所有权值矩阵  $\mathbf{W}$  的 mask (即  $\mathbf{T}$ ) 的列表
11         self.masks = []
12         # 将  $\alpha$ 、 $\beta$ 、 $\gamma$  默认初始化为 0.0001、1、1
13         if alpha is None:
14             self.alpha = 1e-4
15         if beta is None:
16             self.beta = 1
17         if gamma is None:
18             self.gamma = 1
19
20         # 定义辅助剪枝的方法
21         # 在当前这个简易版本中, 只需返回  $w$  和  $w$  abs 即可
22         # 其中  $w$  代指目标权值矩阵、 $w$  abs 代指该权值矩阵的绝对值
23         @staticmethod
24         def get w info(w):
25             with tf.name scope("get w info"):
26                 w abs = tf.abs(w)
27                 return w, w abs
28
29         # 定义执行剪枝的方法
30         # 可以看到, 它会利用到上一个方法的返回值
31         def prune w(self, w, w abs):
32             with tf.name scope("Prune"):
33                 # 依公式计算  $\tilde{\mathbf{T}}$ 
34                 log w = tf.log(tf.maximum(
35                     self.eps, w abs / (w abs mean * self.gamma)))
36                 if self.r > 0:
37                     log_w = tf.minimum(self.r, self.beta * log_w)
38                 # 依公式计算  $\mathbf{T}$ 
39                 mask = tf.maximum(self.alpha / self.beta * log_w, log_w)
40                 self.masks.append(mask)
41                 # 返回  $\mathbf{W}$  和  $\mathbf{T}$  的 Hadamard 乘积  $\tilde{\mathbf{W}}$ 
42                 return w * mask

```

可以看到, 单从实现来说, 软剪枝要比 Surgery 简单许多。Surgery 的相应代码能够在上面提供的链接中获得, 笔者本人是运用了 `tf.cond` 和 Python 的闭包之后才完成了实现, 所以还是有一定难度的。反观软剪枝的话, 它的实现就只是对着公式调用 TensorFlow 的相应函数而已。

而在实际应用软剪枝时, 由于剪枝的对象是权值矩阵, 所以我们需要在线性映射的过程中引入剪枝技术, 这也正是为何我们在 3.5.2 节的最后说自定义线性映射非常重要的原因。

```

01     """
02     实现 AdvancedNN 中的线性映射

```

```

03     net: 线性映射的输入
04     shape: 权值矩阵的 shape
05     appendix: 线性映射这一套运算步骤所属的 name scope 的后缀
06     """
07     def fully_connected_linear(self, net, shape, appendix):
08         with tf.name_scope("Linear{}".format(appendix)):
09             w = init_w(shape, "W{}".format(appendix))
10             # 如果使用剪枝技术的话, 就调用相应方法剪枝
11             if self.pruner is not None:
12                 w = self.pruner.prune_w(*self.pruner.get_w_info(w))
13             # 其余部分和先前的实现是一致的
14             b = init_b([shape[1]], "b{}".format(appendix))
15             self.ws.append(w)
16             self.bs.append(b)
17             return tf.add(
18                 tf.matmul(net, w), b,
19                 name="Linear{}_Output".format(appendix))

```

注意: 该方法是定义在 AdvancedNN 内部的方法, 它虽然和 5.3.2 节中实现 DNDF 时用到的公用函数 `fully_connected_linear` 非常像, 但是地位是完全不一样的。

以上就是对软剪枝技术的简要介绍, 我们在此先做一个小结, 然后会在下一节中给出一些公式推导与实验结果来说明剪枝技术的具体内涵:

- 比起 Surgery 而言, 软剪枝技术损失的信息会少一些, 因为它有一个过渡的过程。
- 实现、调参简便, 且通过调参可以逼近硬剪枝 ($\alpha \rightarrow 0, \beta \rightarrow \infty, \gamma = r = 1$)。
- 虽然实现简便, 但由于用到的公式比较复杂, 所以运行速度不及硬剪枝。
- 一般而言不会达到真正的稀疏性, 所以无法压缩模型。

5.4.4* 软剪枝的算法内涵

我们之前曾经说过, Surgery 不仅能够将已有的权值剪掉, 还能将已被剪掉的权值恢复, 这个性质对于软剪枝来说也是拥有的。不过需要指出的是, 这个“恢复”的过程其实相当复杂。事实上, 如果我们假设 $\gamma = 1$ (即软剪枝的阈值就是 μ) 的话, 就可以发现:

- 已被剪掉的权值如果相对于 μ 很小, 那么它就会尝试回到一个相对于 μ 而言较为适中的大小。
- 已被剪掉的权值通常不是通过将自身变得比 μ 大来恢复, 而是通过 μ 本身变小来恢复的。
- 刚被恢复的权值可能会有留在 μ 的附近的趋势。

注意: 本节中的公式比较长, 但这不是因为它们本身的难度所致, 而是因为软剪枝算法对应的公式比较烦琐所致。因此即使略过大多数的推导也完全没有关系, 只需对上述的三个性质有所体会, 并看一看各个实验结果图即可。

具体而言, 假设要应用软剪枝技术的权值矩阵为 $\mathbf{W}^{(l)}$, 那么它的 `mask` 即为

$$T_{ij} = \max\left(\frac{\alpha}{\beta} \cdot \tilde{T}_{ij}, \tilde{T}_{ij}\right)$$

那么实际应用到神经网络中的权值矩阵 $\widetilde{\mathbf{W}}^{(l)}$ 即为

$$\widetilde{\mathbf{W}}^{(l)} = \mathbf{W}^{(l)} \odot \mathbf{T}$$

而在利用反向传播算法计算梯度时, 由第 3 章的讨论可知

$$\nabla_{\mathbf{W}^{(l)}} L = \boldsymbol{\delta}^{(l)} \times \mathbf{o}^{(l-1)\mathrm{T}}$$

且由链式法则可知

$$\nabla_{\mathbf{W}^{(l)}} L = \sum_{i=1}^{n_l} \sum_{j=1}^{n_{l-1}} \frac{\partial L}{\partial \widetilde{W}_{ij}^{(l)}} \cdot \nabla_{\mathbf{W}} \widetilde{W}_{ij}^{(l)}$$

注意, $\widetilde{W}_{ij}^{(l)}$ 仅受 W_{ij} 的影响, 且

$$\frac{\partial \widetilde{W}_{ij}^{(l)}}{\partial W_{ij}} = \frac{\partial W_{ij} \cdot T_{ij}}{\partial W_{ij}} = T_{ij} + W_{ij} \cdot \frac{\partial T_{ij}}{\partial W_{ij}}$$

再由前文的讨论可知, 当软剪枝中的超参数满足 $0 < \alpha < \beta$ 时, 有

$$T_{ij} = \alpha \log \left(\max \left(\epsilon, \frac{|W_{ij}|}{\mu} \right) \right), \quad \text{if } |W_{ij}| \leq \mu$$

考虑到

$$\begin{aligned} \frac{\partial \log \frac{|W_{ij}|}{\mu}}{\partial W_{ij}} &= \begin{cases} \frac{\partial \log \left(-\frac{W_{ij}}{\mu} \right)}{\partial W_{ij}} & , \quad \text{if } W_{ij} < 0 \\ \frac{\partial \log \frac{W_{ij}}{\mu}}{\partial W_{ij}} & , \quad \text{if } W_{ij} \geq 0 \end{cases} \\ &= \begin{cases} -\frac{1}{W_{ij}} & , \quad \text{if } W_{ij} < 0 \\ \frac{1}{W_{ij}} & , \quad \text{if } W_{ij} \geq 0 \end{cases} = \frac{1}{|W_{ij}|} \end{aligned}$$

从而当 W_{ij} 会被软剪枝裁剪掉 (即 $|W_{ij}| \leq \mu$) 时, 就有

$$\frac{\partial \widetilde{W}_{ij}^{(l)}}{\partial W_{ij}} = \begin{cases} \alpha \log \epsilon & , \quad \text{if } |W_{ij}| \leq \epsilon \cdot \mu \\ \alpha \left(\log \frac{|W_{ij}|}{\mu} + \frac{W_{ij}}{|W_{ij}|} \right) & , \quad \text{if } |W_{ij}| > \epsilon \cdot \mu \end{cases}$$

于是结合

$$\frac{W_{ij}}{|W_{ij}|} = \text{sign}(W_{ij})$$

可知，被软剪枝裁剪后的 W_{ij} 的梯度为

$$(\nabla_{\mathbf{W}^{(l)}L})_{ij} = \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}} \cdot \frac{\partial \tilde{W}_{ij}^{(l)}}{\partial W_{ij}} = \begin{cases} \alpha \log \epsilon \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}} & , \text{ if } |W_{ij}| \leq \epsilon \cdot \mu \\ \alpha \left(\log \frac{|W_{ij}|}{\mu} + \text{sign}(W_{ij}) \right) \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}} & , \text{ if } |W_{ij}| > \epsilon \cdot \mu \end{cases}$$

其中

$$\frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}} = (\boldsymbol{\delta}^{(l)} \times \mathbf{o}^{(l-1)\text{T}})_{ij}$$

由于 α 通常会取一个很小的值（比如说 0.0001），所以该公式就能佐证本节开头所提到的两个性质。具体而言：

- 当被剪掉的 W_{ij} 的绝对值相对于 μ 很小（即 $|W_{ij}| \leq \epsilon \cdot \mu$ ）时，就有

$$(\nabla_{\mathbf{W}^{(l)}L})_{ij} \approx \alpha \log \epsilon \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}}$$

这意味着当 ϵ 取适当的值时， $(\nabla_{\mathbf{W}^{(l)}L})_{ij}$ 将会是一个适中的数。比如在 5.4.3 节所提供的实现中，有

$$\alpha = 10^{-4}, \quad \epsilon = 10^{-12}$$

那么此时就有

$$\alpha \log \epsilon = -2.7 \cdot 10^{-3}$$

即虽然相应的梯度确实削弱了很多，但是毕竟没有完全消失。多迭代几次，那么就可以期望 W_{ij} 回到相对于 μ 而言较为适中的大小。

- 当被剪掉的 W_{ij} 的绝对值离 μ 比较近时，由于此时

$$|W_{ij}| > \epsilon \cdot \mu \wedge \frac{|W_{ij}|}{\mu} \approx 1$$

从而就有

$$\alpha \left(\log \frac{|W_{ij}|}{\mu} + \text{sign}(W_{ij}) \right) \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}} \approx \alpha \cdot \text{sign}(W_{ij}) \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}}$$

这意味着此时 W_{ij} 的梯度大概只是第一种情况（ $|W_{ij}| \leq \epsilon \cdot \mu$ ）中的梯度的 $1 / \log \epsilon$ （约 0.036），即它将会是一个很小的数，从而 W_{ij} 在梯度下降法中就不会被更新。此时如果 W_{ij} 想要恢复的话，由于它自身无法改变，所以就只能通过 μ 本身变小的方式使它自身恢复。

注意： μ 本身变小通常表现为其余没被剪枝的权值的绝对值变小。同时，由于 Surgery 裁剪过的权值将恒为 0，所以一旦 W_{ij} 被 Surgery 裁剪了，那么它就会严格不被更新，这就意味着对于 Surgery 而言，除非其余没被剪枝的权值的绝对值均值变小或整体的绝对值标准差变大，否则 W_{ij} 将会一直不变， \tilde{W}_{ij} 也将一直为 0。

这两个性质其实就是本节开头所说的前两个性质。此外，如果 W_{ij} 刚从被软剪枝裁剪的状态下恢复的话，它就会满足

$$|W_{ij}| > \mu \wedge \frac{|W_{ij}|}{\mu} \approx 1$$

那么由前文的讨论可知，此时有

$$\tilde{T}_{ij} = \min\left(r, \beta \log \frac{|W_{ij}|}{\mu}\right) = \beta \log \frac{|W_{ij}|}{\mu} > 0$$

于是

$$T_{ij} = \max\left(\frac{\alpha}{\beta} \cdot \tilde{T}_{ij}, \tilde{T}_{ij}\right) = \beta \log \frac{|W_{ij}|}{\mu}$$

即

$$\frac{\partial \tilde{W}_{ij}^{(l)}}{\partial W_{ij}} = \beta \left(\log \frac{|W_{ij}|}{\mu} + \text{sign}(W_{ij}) \right) \approx \beta \cdot \text{sign}(W_{ij})$$

从而

$$\frac{\partial L}{\partial W_{ij}} = \beta \cdot \text{sign}(W_{ij}) \cdot \frac{\partial L}{\partial \tilde{W}_{ij}}$$

于是不难看出，只需要满足

$$\frac{\partial L}{\partial \tilde{W}_{ij}} > 0$$

的话，那么当 W_{ij} 为正时其导数亦为正， W_{ij} 为负时其导数亦为负，从而在进行梯度下降时， W_{ij} 的绝对值就总是会变小。又由于从概率的角度来看，在没有任何先验知识时可以认为 \tilde{W}_{ij} 有一半的概率为正，从而 W_{ij} 从被裁剪的状态恢复之后，其绝对值会有一半的概率再次下降，即 W_{ij} 可能又会被裁剪，或说至少会留在 μ 的附近。这就在一定程度上佐证了本节开头所说的第三个性质。

注意我们曾在 5.4.2 节的最后给出过“绝对值权值越大则相应的连接越重要”这个说法。如果我们承认这个说法的话，那么上述三个性质就可以给软剪枝一个直观的解释：当某个权值 W_{ij} 被裁剪后，或者离阈值 μ 不远时，软剪枝都可以让 W_{ij} 处于“待命”的状态——它在相对于 μ 较小时会尝试回到一个适中的位置，而在接近 μ 时则会“按兵不动”。此时，只要其余权值的绝对值变小一些（即重要性降低一些），由于 W_{ij} 通常处于“蓄势待发”的位置，那么 W_{ij} 就会有比较大的概率恢复回来（即它的重要性就会恢复一些）；同理，只要其余权值的绝对值变大一些（即重要性提升一些），由于有许多刚恢复回来的 W_{ij} 仍在离 μ 不远的地方“驻留”，所以这些 W_{ij} 就会有比较大的概率又被剪掉（即它的重要性就会被抑制一些）。反观 Surgery， W_{ij} 如果本来很小的话，它不会像软剪枝那样能够回到相对于 μ 较为适中的大小，而是一直保持在很小的地方，从而给 W_{ij} 的恢复带来极大的困难（即它可能就会一直是不重要的权值）；而当 W_{ij} 恢复回来之后，它就会马上和离阈值 μ 很远的权值的地位等同，从而它也很难再被剪掉（即它

可能从此就一直很重要)。

不过, 这些说法毕竟是基于“权值绝对值越大则越重要”的理念的。如果不对该理念进行一定程度的证明的话, 整套体系就会站不住脚。为此, 我们会在本节最后给出一个特殊情况下的证明, 这里就暂时先承认这个说法的正确性。

总之, 软剪枝的核心思想就在于“过渡”二字。它在从正常状态到被剪掉的过程中有过渡, 在恢复的过程中也会有过渡。如果用重要性来进行叙述的话, 那么软剪枝的核心思想就在于给权值重要性的变化引入了很合理的过渡过程。不过, 这些东西仅仅只是(稍微带一点理论证明的)直观阐述, 为了让这些说法更具说服力, 我们可以通过具体的实验来进行佐证。假设现在要对 W 应用剪枝技术, 那么就可以定义如下几个指标来佐证前文提及的软剪枝的三个性质。

- **pruned_ratio**: 当前处于被裁剪的权值所占的比例, 简称为 **pr**。
- **recovered_ratio**: 一开始被裁剪的权值在当前得到恢复的比例, 简称为 **recr**。
- **running_recovered**: 第 $t-1$ 步迭代中被裁剪的权值在第 t 步中得以恢复的个数, 简称为 **runr**。
- **pruned_w_abs_residual**: 第 $t-1$ 步迭代中被裁剪的权值 W_{ij} 在第 t 步中的变化, 简称为 **pwar**。

实验所采用的数据集则是我们曾经在第3章中用过的线性噪声数据集的高维版本, 它的生成规则和第3章介绍的二维版本的生成规则是类似的:

- 使用一维标准正态分布来采样出原始的 n 维特征向量 \mathbf{x} 。
- 在 \mathbf{x} 的基础上, 加上服从均值为 0、标准差为 0.5 的正态分布噪声, 得到训练所用的特征向量 $\tilde{\mathbf{x}}$ 。
- 在 n 个维度 $(x^{(1)}, x^{(2)}, \dots, x^{(n)})$ 中挑选出 k 个 $(x^{(i_1)}, x^{(i_2)}, \dots, x^{(i_k)})$ 作为真正有用的特征。
- 随机初始化权值向量 $\mathbf{w} = (w^{(1)}, w^{(2)}, \dots, w^{(k)})^T$, 并根据

$$\sum_{j=1}^k w^{(j)} x^{(i_j)}$$

是否大于 0 来给相应的带噪声的特征向量 $\tilde{\mathbf{x}}$ 打上标签 y :

$$y = \begin{cases} 1 & , \text{ if } \sum_{j=1}^k w^{(j)} x^{(i_j)} > 0 \\ 0 & , \text{ if } \sum_{j=1}^k w^{(j)} x^{(i_j)} \leq 0 \end{cases}$$

这就得到了我们的训练集:

$$D = \{(\tilde{\mathbf{x}}_1, y_1), (\tilde{\mathbf{x}}_2, y_2), \dots, (\tilde{\mathbf{x}}_N, y_N)\}$$

- 至于测试集, 则是不带噪声的特征向量与标签的组合。

可以看到，这个高维线性噪声数据集不仅在训练集中加入了噪声，还加入了许多冗余的特征。由于在做实验时通常会取 $n = 100$ 以及 $k = 5$ ，这就意味着有 95% 的特征都是冗余的，而这种高冗余度恰恰能够比较好地检验剪枝技术的有效性。

该高维线性噪声数据集的生成代码如下所示。

```
01 """
02     生成高维线性噪声数据集
03     size: 数据集样本数的规模
04     n dim: 总特征维数 (n)
05     n valid: 有效的特征维数 (m)
06     noise scale: 噪声的强度
07     test ratio: 测试集的规模相比总规模的比例
08 """
09 def gen_noisy_linear(size=10000, n_dim=100, n_valid=5,
10     noise_scale=0.5, test_ratio=0.15):
11     # 生成原始的 n 维特征向量 x
12     x_train = np.random.randn(size, n_dim)
13     # 加上噪声
14     x_train_noise = x_train + np.random.randn(size, n_dim) * noise_scale
15     # 生成测试集
16     x_test = np.random.randn(int(size*test_ratio), n_dim)
17     # 随机挑选出 m 个有效特征
18     idx = np.random.permutation(n_dim)[:n_valid]
19     # 生成权值矩阵
20     w = np.random.randn(n_valid, 1)
21     # 生成训练集、测试集的标签
22     y_train = (x_train[..., idx].dot(w) > 0).astype(np.int8).ravel()
23     y_test = (x_test[..., idx].dot(w) > 0).astype(np.int8).ravel()
24     # 返回训练集和测试集
25     return (x_train_noise, y_train), (x_test, y_test)
```

在此基础上，我们把神经网络的各种超参数定义成：

- 训练步长 (n_epoch) 为 40。
- 不使用 Wide 模型与 DNDF，但使用剪枝技术。同时在剪枝技术中，软剪枝的超参数均采用默认参数 ($\alpha = 10^{-4}, \beta = \gamma = r = 1, \epsilon = 10^{-12}$)，Surgery 则采用一直剪枝的方式而非像论文所说的那样依概率剪枝。
- 网络结构为两层神经网络，其中隐藏层有 512 个神经元。

那么相应的核心测试代码如下所示（虽然我们还没说明怎么定义结构超参数，不过如果单看调用的话，由于相应的代码非常直观，所以大抵是不会引起理解上的障碍的）。

```
01 # 生成高维线性噪声数据集
02 from Util.Util import DataUtil
03 (x, y), (x_test, y_test) = DataUtil.gen_noisy_linear(one_hot=False)
04
05 # 定义网络结构
06 nn = Advanced(
```

```

07 # 将此时 AdvancedNN 的名字定义为 NoisyLinear
08 "NoisyLinear",
09 # 初始化数据超参数
10 # 由于特征向量由 100 个连续特征组成，而且是分类问题，所以：
11 # - numerical idx 应该设置为 [True] * 100 + [False]
12 # - categorical columns 应该留空
13 data_info={
14     "numerical idx": [True] * 100 + [False],
15     "categorical columns": []
16 },
17 # 初始化模型超参数
18 # - 将训练步数设置为 40
19 model_param_settings={
20     "n_epoch": 40
21 },
22 # 初始化结构超参数
23 # - 设置为不使用 Wide 模型
24 # 由于实际实现时只会对 Wide 模型使用 DNDF（原因会在下一节说明）
25 # 所以不使用 Wide 模型就意味着不使用 DNDF
26 # - 设置为使用软剪枝技术
27 # 在用 Surgery 进行对比时，只需把 prune method 对应的
28 # soft prune 改成 surgery 即可
29 # - 把隐藏层设置为只有一个，且相应的神经元个数为 512
30 model_structure_settings={
31     "use wide network": False,
32     "use pruner": True,
33     "pruner params": {"prune method": "soft prune"},
34     "hidden units": [512]
35 }
36 # 调用 fit 方法进行训练
37 ).fit(x, y, x_test, y_test, snapshot_ratio=0)

```

为了保证实验的可靠性，接下来展示的结果都是多次重复实验后的、具有普遍意义的结果。同时，由于各个指标之间的取值范围与量纲各不相同，所以为了能够直观地看出各个指标之间在训练过程中的相对关系，我们需要对各个指标做标准化（减均值、除标准差）处理。

下面就从最直观的两个指标——pruned_ratio (pr) 和 recovered_ratio (recr) 与 w_abs_mean (mu) 的相对关系开始展示（如图 5.8 和图 5.9 所示，其中横、纵坐标分别是迭代步数与相对大小，后同）。¹由于这两个指标很关键，所以在讨论这两个指标时，我们会自然地引入其他指标的讨论。

从中可以看出许多软剪枝的有趣的性质。比如，虽然软剪枝和 Surgery 的 pr 都在不断提升，但是软剪枝的 pr 在提升的过程中会有很多下穿的部分，不像 Surgery 那般平滑，这就说明软剪枝的动态恢复能力比 Surgery 要强很多。这一点通过直接比较它们的 running_recovered (runr) 与 pruned_w_abs_residual (pwar) 将会更加明显。

¹ 编者注：本章的彩色图片可从 <http://www.broadview.com.cn/34238> 下载进行观看。

先来看 `runr`（如图 5.10、图 5.11 所示），可以发现对于软剪枝而言，当前迭代中被剪掉的权值有很多都会在下一步迭代中恢复回来，但对于 `Surgery` 来说则是不存在的事。

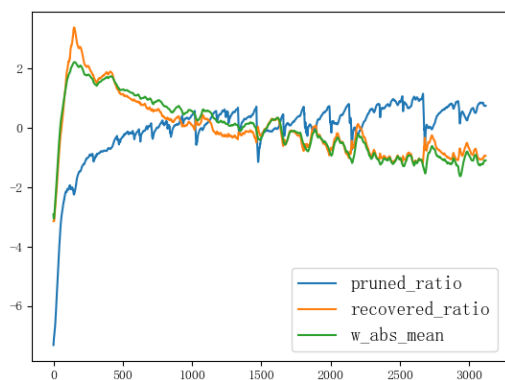


图5.8 软剪枝下，pr、recr和mu之间的关系

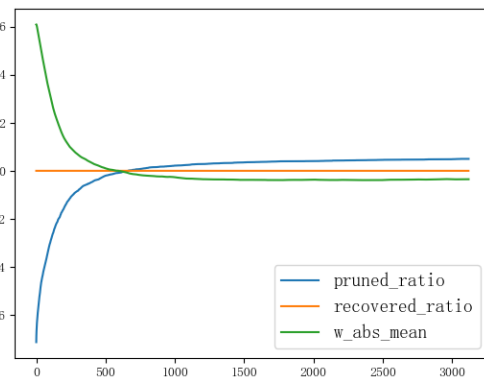


图5.9 Surgery下，pr、recr和mu之间的关系

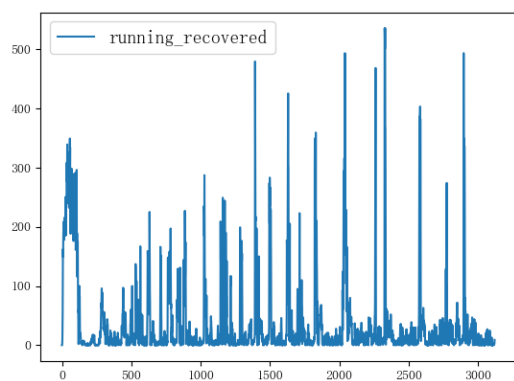


图5.10 软剪枝下的runr

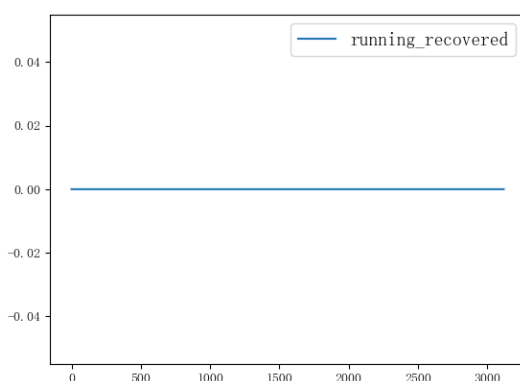


图5.11 Surgery下的runr

再来看 `pwar`（如图 5.12、图 5.13 所示），可以发现对于软剪枝而言，当前迭代中被剪掉的权值 w_{ij} 在下一步迭代中的变化量一直都在一个比较大的区间内波动（ 10^{-3} 量级），而 `Surgery` 除了在一开始算法尚未稳定时有一个小的尖峰（ 10^{-6} 量级）以外，剩下的时间内 `pwar` 都几乎为 0。

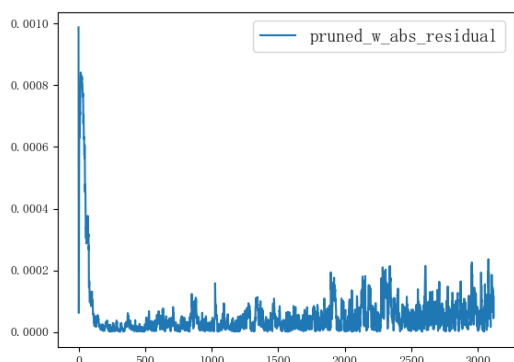


图5.12 软剪枝下的pwar

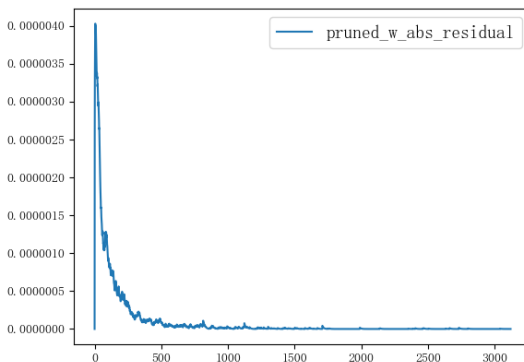


图5.13 Surgery下的pwar

综上所述，软剪枝的动态恢复能力确实比 Surgery 要好很多。不过需要指出的是，由于 Surgery 的剪枝阈值 $(0.9 \cdot (\mu + \beta \text{std}))$ 在权值波动比较大（即 std 比较大）时通常会比软剪枝的剪枝阈值 (μ) 大，所以 Surgery 的 pr 会比软剪枝的 pr 大，如图 5.14 所示。

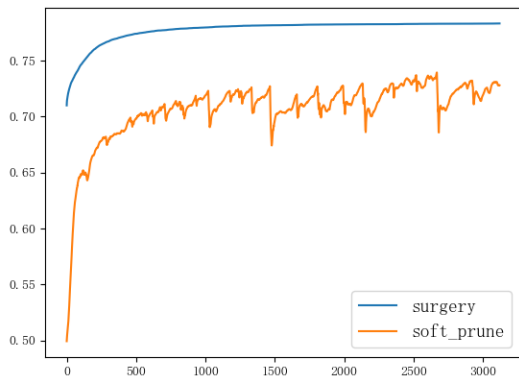


图5.14 Surgery与软剪枝的pr

同时值得指出的是，从图 5.8、图 5.9 中还可以看出一个非常有意思的现象：软剪枝的 recovered_ratio (recr) 的变化是和 w_abs_mean (μ) 的变化呈强正相关性的，这恰恰在一定程度上说明了会有许多权值在阈值 μ 附近处于“待命”的状态。我们可以分两个方面来梳理这里的逻辑。

- 第一个方面是为何 recr 的变化与 μ 的变化呈强相关。这主要是因为恢复比率 (recr) 中贡献最大的往往是阈值 (μ) 附近的部分，所以如果有许多权值在 μ 附近待命的话， μ 本身的变化自然就会在极大程度上影响 recr 。为了更有说服力，我们可以画出各种情况下（使用 Surgery、使用软剪枝和不使用剪枝）处于 μ 附近的权值占所有权值的比例随训练过程的变化来佐证这一点，如图 5.15 所示，其中所谓的“ μ 附近的权值”，指的是满足

$$\mu - \text{std} \leq W_{ij} \leq \mu + \text{std}$$

的权值 W_{ij} 。从图 5.15 中可以看到，使用 Surgery 之后，处于 μ 附近的权值比例会比不使用剪枝时要小，这主要是因为被 Surgery 裁剪的权值将会一直保持在原地不动，所以对于初始化时远小于 $\mu + \text{std}$ 的权值，它们就会一直保持小于 $\mu + \text{std}$ 的状态。而软剪枝由于有前文所提及的诸多性质，所以使用软剪枝之后，在 μ 附近待命的权值比例相对而言会非常高。

此外，如果我们只关心一开始被裁剪掉的（即小于 μ 的）权值在 μ 附近的比例的话，这个性质就会更明显了，如图 5.16 所示。可以看到，这些权值会在训练的初期迅速集中在 μ 的附近（比例高达 95%），然后在训练过程中也都基本会留在 μ 的附近。

- 第二个方面就是为何 recr 的变化与 μ 的变化呈正相关。这毫无疑问是反直觉的，因为从直观上来说， μ 越大则权值应该越难恢复才对。而事实上，该性质确实和“软剪枝下权值大多集中在 μ 附近”这个性质不同，它并不是一个普遍的性质。

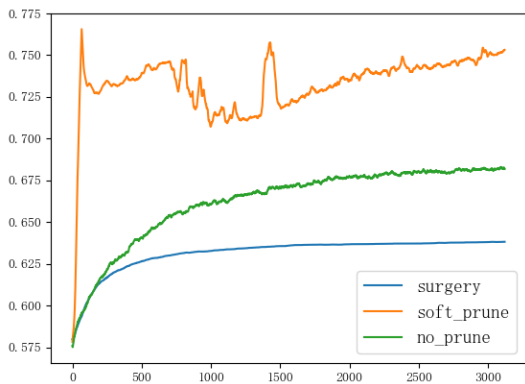


图5.15 使用Surgery、使用软剪枝和不使用剪枝下的处于 μ 附近的权值的比例

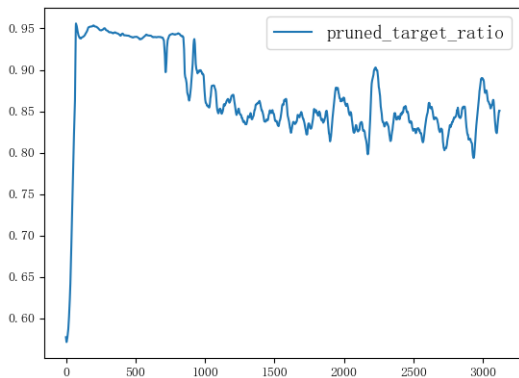


图5.16 使用软剪枝时，一开始小于 μ 的权值在训练过程中处于 μ 附近的比例

不过，结合“权值大多在 $(\mu - \text{std}, \mu + \text{std})$ 区间里面”的这个普遍性质，我们确实可以对图 5.8 中的正相关现象做出解释。具体而言，由前文知软剪枝后的 W_{ij} 在 μ 附近的梯度约为

$$\frac{\partial L}{\partial W_{ij}} = \alpha \cdot \text{sign}(W_{ij}) \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}}$$

在绝对值的意义下，就有

$$\frac{\partial L}{\partial |W_{ij}|} = \alpha \cdot |\text{sign}(W_{ij})| \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}} = \alpha \cdot \frac{\partial L}{\partial \tilde{W}_{ij}^{(l)}}$$

即 $|W_{ij}|$ 的梯度和 \tilde{W}_{ij} 的梯度呈正相关关系。从而当 \tilde{W}_{ij} 的绝对值均值发生变化时，只要 \tilde{W}_{ij} 和 W_{ij} 的符号不要有一一对应关系，那么就总有一部分本来就处于 μ 附近的 W_{ij} 能够得到相应的变化，从而超出（或进入） μ 的范围。

当然该解释也能看出，像图 5.8 所示的那么强的正相关性只是这单一的数据集所导致的，对于其他数据集而言，正相关性可能就会弱一些，甚至不复存在。比如，我们同样使用高维噪声线性数据集，但是把 n 设置为 10 的话， recr 和 μ 的关系就将如图 5.17 所示。可以看到，虽然在 2000 步之前还呈比较强的正相关性，可是在 2000 步之后就呈负相关性了。

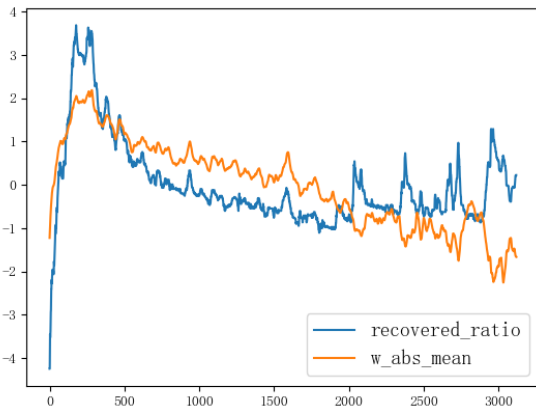


图5.17 $n = 10, m = 5$ 的高维线性数据集上 recr 和 μ 之间的关系

注意：即使 recr 和 mu 不一定呈正相关性，但由于权值会聚集在 mu 附近的这个性质是通用的性质，所以相关性是一定存在的。

以上，我们就较为细致地讨论了各种指标的方方面面。而我们曾在 5.4.2 节中说过，软剪枝在通过调整超参数 ($\alpha \rightarrow 0, \beta \rightarrow \infty$) 之后是能逼近 Surgery 的表现的，所以下面我们就对调整超参数后的软剪枝算法也做一遍相应的实验。具体而言，我们会取

$$\alpha = 10^{-8}, \quad \beta = 10^8$$

由于调整了超参数之后的软剪枝算法也变得比较“硬”了，所以我们简称它为“硬剪枝 (hard_prune)”算法。

首先来看在硬剪枝算法下， pruned_ratio 、 recovered_ratio 和 w_abs_mean 之间的关系，如图 5.18 所示。可以看到，硬剪枝的表现与图 5.8 中软剪枝的表现相去甚远，但和图 5.9 中 Surgery 的表现非常类似。

然后是 running_recovered ，硬剪枝在该指标下的表现如图 5.19 所示。可以看到，虽然它不像 Surgery 那样是一条恒为 0 的直线，但是大多数取值都是 1，而且最大的取值也只是 6，总体而言还是很贴近的。

最后是 $\text{pruned_w_abs_residual}$ ，如图 5.20 所示，可以看到比起图 5.12 中软剪枝的表现，硬剪枝同样是和图 5.13 中 Surgery 的表现类似。

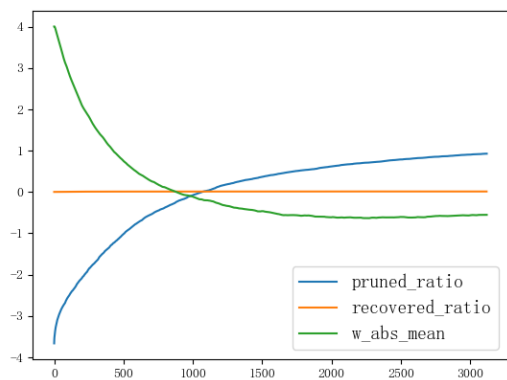


图 5.18 硬剪枝下， pr 、 recr 和 mu 之间的关系

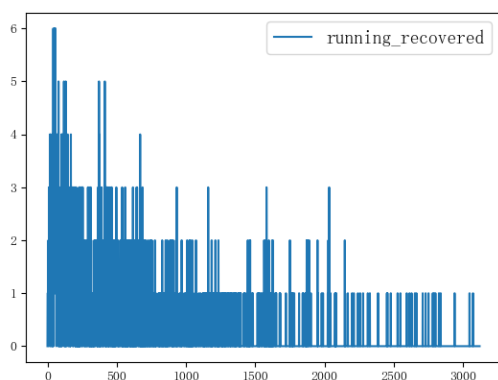


图 5.19 硬剪枝下的 runnr

总之我们说明了，硬剪枝确实能在一定程度上逼近 Surgery 的表现。不过也正如前文所说，Surgery 的剪枝阈值在权值波动比较大时通常会比软剪枝的剪枝阈值 (mu) 大，所以对于硬剪枝而言，它的 pr 仍会比 Surgery 的 pr 要小。此外，由于初始化时比较小的权值无法像软剪枝那样迅速地集中在 mu 附近，所以它的 pr 会比软剪枝的 pr 还小，如图 5.21 所示。

注意：从性质上来看的话，硬剪枝的表现和 Surgery 的表现是一致的：它们都没有软剪枝那“下穿”的部分，即它们的动态恢复能力都很差（或说几乎没有）。

此外，笔者将本节所涉及的所有测试专用代码放在了 GitHub 上，感兴趣的读者可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/Pruner/Test.py。

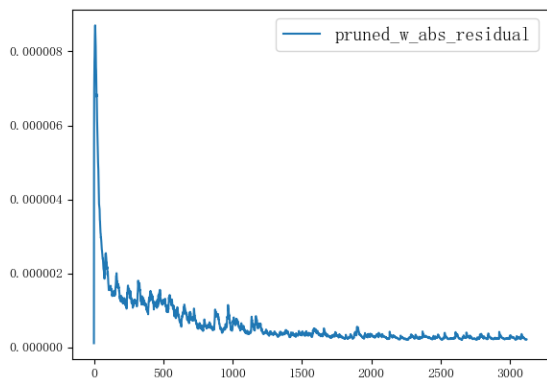


图5.20 硬剪枝下的pwar

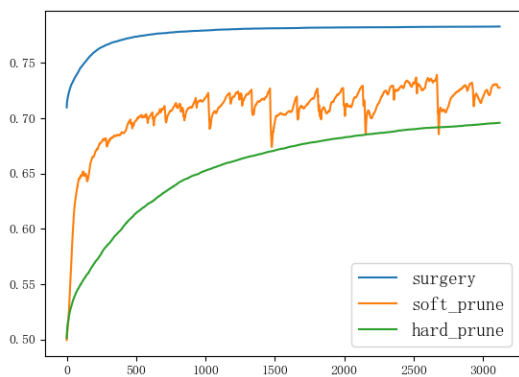


图5.21 Surgery、软剪枝与硬剪枝的pr

在本节的最后，我们来提供“权值绝对值越大则重要性越大”这个说法的证明。需要指出的是，该证明做了比较多的强假设，相关的推导也比较烦琐，所以如果大家能够直接接受这个说法的话，即使不看接下来的推导也没有太大的关系。

如果确实感兴趣的话，则可以先来看看该证明所针对的特殊情况：

- 使用的数据集为高维噪声线性数据集。
- 使用的神经网络是退化后的神经网络（即线性模型）。
- 输出层使用的组合是 Sigmoid+Cross Entropy（即输出层只有 1 个神经元）。
- 训练方式为 Vanilla Update，训练算法为 Batch Gradient Descent，训练速率为 η 。

那么此时可以证明的是，对于（唯一的）权值矩阵 $\mathbf{W}^{(1)}$ （注意由于输出层只有 1 个神经元，所以 $\mathbf{W}^{(1)}$ 严格来说不是一个矩阵，而是一个行向量）而言，绝对值越大的权值确实就越重要。具体而言，由第 3 章的公式可知 $\mathbf{W}^{(1)}$ 的梯度为

$$\nabla_{\mathbf{W}^{(1)}} L = \boldsymbol{\delta}^{(1)} \mathbf{o}^{(0)\top}$$

由于我们没有做任何数据预处理（而且对于高维噪声线性数据集而言也不需要做），所以 $\mathbf{o}^{(0)}$ 就是原始特征：

$$\mathbf{o}^{(0)} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^\top$$

不失一般性，不妨假设前 k 个特征 $x^{(1)} \sim x^{(k)}$ 是真正有用的特征，并分别简记这 k 个特征与剩下的 $n - k$ 个特征为

$$\hat{\mathbf{x}} \triangleq (x^{(1)}, \dots, x^{(k)})^\top, \quad \mathbf{x}^\# = (x^{(k+1)}, \dots, x^{(n)})^\top$$

则 $\mathbf{o}^{(0)}$ 就可以简记为

$$\mathbf{o}^{(0)} = \begin{bmatrix} \hat{\mathbf{x}} \\ \mathbf{x}^\# \end{bmatrix}$$

且数据集标签的生成规则可以简记为

$$y = \begin{cases} 1 & , \text{ if } \mathbf{w}^\top \hat{\mathbf{x}} > 0 \\ 0 & , \text{ if } \mathbf{w}^\top \hat{\mathbf{x}} \leq 0 \end{cases}$$

来作为神经网络拟合的目标，其中 \mathbf{w}^T 是行向量 $\mathbf{W}^{(1)}$ 的前5个元素。此外，由于我们使用了退化后的神经网络，且输出层使用了 Sigmoid+Cross Entropy，所以局部梯度 $\delta^{(1)}$ 将会是一个标量，其数值为

$$\delta^{(1)} = o^{(1)} - y = \mathbf{W}_0^{(1)} + \mathbf{W}^{(1)}\mathbf{o}^{(0)} - y$$

从而

$$\begin{aligned}\nabla_{\mathbf{W}^{(1)}}L &= (\mathbf{W}_0^{(1)} + \mathbf{W}^{(1)}\mathbf{o}^{(0)} - y)\mathbf{o}^{(0)T} \\ &= \mathbf{W}_0^{(1)}\mathbf{o}^{(0)T} + \mathbf{W}^{(1)}\mathbf{o}^{(0)}\mathbf{o}^{(0)T} - y\mathbf{o}^{(0)T}\end{aligned}$$

注意到 $\mathbf{x}^{(1)} \sim \mathbf{x}^{(n)}$ 都服从均值为0的正态分布，所以 $\mathbf{o}^{(0)}$ 的期望也将是0，从而

$$\begin{aligned}E(\nabla_{\mathbf{W}^{(1)}}L) &= E(\mathbf{W}_0^{(1)}\mathbf{o}^{(0)T} + \mathbf{W}^{(1)}\mathbf{o}^{(0)}\mathbf{o}^{(0)T} - y\mathbf{o}^{(0)T}) \\ &= E(\mathbf{W}_0^{(1)}\mathbf{o}^{(0)T}) + E(\mathbf{W}^{(1)}\mathbf{o}^{(0)}\mathbf{o}^{(0)T}) - E(y\mathbf{o}^{(0)T}) \\ &= \mathbf{W}_0^{(1)}E(\mathbf{o}^{(0)T}) + \mathbf{W}^{(1)}E(\mathbf{o}^{(0)}\mathbf{o}^{(0)T}) - E(y\mathbf{o}^{(0)T}) \\ &= \mathbf{W}^{(1)}\text{cov}(\mathbf{o}^{(0)}) - E(y\mathbf{o}^{(0)T})\end{aligned}$$

其中，由于 $E(\mathbf{o}^{(0)}) = 0$ ，所以 $\text{cov}(\mathbf{o}^{(0)}) = E(\mathbf{o}^{(0)}\mathbf{o}^{(0)T})$ 所代指的正是 $\mathbf{o}^{(0)}$ 的协方差矩阵。此外，由于原始数据是在一维标准正态分布中采的样，所以 $\mathbf{o}^{(0)}$ 的协方差矩阵即为单位矩阵：

$$\text{cov}(\mathbf{o}^{(0)}) = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}_{n \times n}$$

于是就有

$$E(\nabla_{\mathbf{W}^{(1)}}L) = \mathbf{W}^{(1)} - E(y\mathbf{o}^{(0)T})$$

注意，我们使用的训练算法是 Batch Gradient Descent，这意味着每次迭代中都会用所有的数据来计算梯度。当训练样本足够多时，由频率估计概率的思想，我们就可以认为用于更新参数的梯度就是梯度的期望。又由于我们使用了训练速率为 η 的 Vanilla Update，所以每次迭代中更新后的 $\mathbf{W}^{(1)}$ 即为

$$\begin{aligned}\mathbf{W}_{\text{new}}^{(1)} &= \mathbf{W}^{(1)} - \eta E(\nabla_{\mathbf{W}^{(1)}}L) \\ &= \mathbf{W}^{(1)} - \eta (\mathbf{W}^{(1)} - E(y\mathbf{o}^{(0)T})) \\ &= (1 - \eta)\mathbf{W}^{(1)} + \eta E(y\mathbf{o}^{(0)T})\end{aligned}$$

注意到

$$y\mathbf{o}^{(0)T} = y(\hat{\mathbf{x}}^T, \mathbf{x}^{\#T}) = (y\hat{\mathbf{x}}^T, y\mathbf{x}^{\#T})$$

从而

$$E(y\mathbf{o}^{(0)T}) = (E(y\hat{\mathbf{x}}^T), E(y\mathbf{x}^{\#T}))$$

又知

$$y = \begin{cases} 1 & , \text{ if } \mathbf{w}^T \hat{\mathbf{x}} > 0 \\ 0 & , \text{ if } \mathbf{w}^T \hat{\mathbf{x}} \leq 0 \end{cases}$$

而且考虑到 $\hat{\mathbf{x}}$ 、 $\mathbf{x}^\#$ 自身各个维度之间也相互独立，所以我们可以将上述两个期望向量拆成一个一个单独的期望值来分析。具体而言，假设 $\mathbf{x}^\# = (x^{\#(1)}, \dots, x^{\#(n-k)})^T$ ，那么就会有

$$E(y\mathbf{x}^{\#T}) = (E(yx^{\#(1)}), \dots, E(yx^{\#(n-k)}))$$

于是结合 $\mathbf{x}^\#$ 的期望为 0，以及 $\hat{\mathbf{x}}$ 与 $\mathbf{x}^\#$ 相互独立，就有：

$$\begin{aligned} E(yx^{\#(i)}) &= \int_{-\infty}^{+\infty} yx^{\#(i)} p(x^{\#(i)}) dx^{\#(i)} \\ &= y \int_{-\infty}^{+\infty} x^{\#(i)} p(x^{\#(i)}) dx^{\#(i)} = 0, \quad i = 1, 2, \dots, n-k \end{aligned}$$

从而

$$E(y\mathbf{x}^{\#T}) = 0$$

类似的，假设 $\hat{\mathbf{x}} = (\hat{x}^{(1)}, \hat{x}^{(2)}, \dots, \hat{x}^{(k)})^T$ ，就有

$$E(y\hat{\mathbf{x}}^T) = (E(y\hat{x}^{(1)}), \dots, E(y\hat{x}^{(k)}))$$

其中

$$E(y\hat{x}^{(i)}) = \int_{-\infty}^{+\infty} y\hat{x}^{(i)} p(\hat{x}^{(i)}) d\hat{x}^{(i)}, \quad i = 1, 2, \dots, k$$

考虑到 y 只有在 $\mathbf{w}^T \hat{\mathbf{x}} > 0$ 时才取 1，且 $w^{(i)}$ 取到 0 点的概率为 0，那么如果令

$$\hat{\mathbf{x}}^{\#(i)} \triangleq -\frac{1}{w^{(i)}} \sum_{j=2}^k w^{(j)} x^{(j)}$$

的话，就有

$$E(y\hat{x}^{(i)}) = \begin{cases} \int_{\hat{\mathbf{x}}^{\#(i)}}^{+\infty} \hat{x}^{(i)} p(\hat{x}^{(i)}) d\hat{x}^{(i)} & , \text{ if } w^{(i)} > 0 \\ \int_{-\infty}^{\hat{\mathbf{x}}^{\#(i)}} \hat{x}^{(i)} p(\hat{x}^{(i)}) d\hat{x}^{(i)} & , \text{ if } w^{(i)} < 0 \end{cases}$$

注意， $\hat{x}^{(i)}$ 是服从标准正态分布的随机变量，所以就有

$$E(y\hat{x}^{(i)}) = \begin{cases} > 0 & , \text{ if } w^{(i)} > 0 \\ < 0 & , \text{ if } w^{(i)} < 0 \end{cases}$$

即

$$E(y\hat{x}^{(i)}) \cdot \text{sign}(w^{(i)}) > 0$$

从而

$$E(y\hat{\mathbf{x}}^T) \odot \text{sign}(\mathbf{w}^T) > 0$$

综上所述，更新后的 $\mathbf{W}^{(1)}$ 就能写成

$$\mathbf{W}_{\text{new}}^{(1)} = (1 - \eta)\mathbf{W}^{(1)} + \eta \cdot ([E(y\hat{\mathbf{x}}^T) \ 0 \ \dots \ 0])$$

且 $E(y\hat{\mathbf{x}}^T)$ 和 $\mathbf{W}^{(1)}$ 中对应的元素 \mathbf{w}^T 同号。这就意味着 $\mathbf{W}^{(1)}$ 在每次更新中除了会有一个整体的、 $1 - \eta$ 倍的衰减以外，对于前 k 个权值而言，原本大于0的元素会变得更大，原本小于0的元素会变得更小。换句话说，对 $\mathbf{W}^{(1)}$ 中真正有用的 k 个权值而言，更新后的权值的绝对值会变大：

$$|\mathbf{w}_{\text{new}}^{(1)}|_i > (1 - \eta)|\mathbf{w}_i^{(1)}|, \quad i = 1, 2, \dots, k$$

而对于其余 $n - k$ 个（没有实际用处的）权值来说，由于相应的梯度为0，所以它们只是单纯地以指数级别的速度衰减，衰减速度的系数为 $(1 - \eta)$ 。这就证明了对于权值而言，其绝对值越大确实意味着它更重要。

5.5 AdvancedNN 的结构设计

在前4节的讨论中，我们依次介绍了 Dropout 与 Batch Normalization 这两个神经网络训练时常常可以使用的技巧、Wide and Deep 与 DNDF 模型，以及神经网络中的剪枝算法。不难看出，这些技术之间相互并不矛盾，甚至还有相当强的互补性。不过，由于 Dropout 和 Batch Normalization 都是泛用型的技巧，所以它们可能和其余技术之间的互补性会差一些。因此在本节的讨论中，我们会默认使用这两个技巧，并不对它们和其余技术的联系做额外的说明；相对应的，我们将会把主要精力放在 WnD、DNDF 与剪枝之间的融合上。

5.5.1 AdvancedNN 的实现补足

在前文介绍诸多技术的实现时，其实我们已经把 AdvancedNN 的实现介绍得差不多了，遗留下来的有两个部分：结构超参数的初始化方法以及 feed_dict 的构建方法，下面我们就展示一下这两者的具体实现代码，如代码 5.4 所示。

代码 5.4 AdvancedNN 的实现：e_AdvancedNN/NN.py

```
01 # 初始化结构超参数
02 def init_model_structure_settings(self):
03     # 注意我们在前文说过，可以只根据样本的数量来
04     # 经验性地定出比较合适的隐藏层信息，所以这里默认取为 None
05     self.hidden_units = self.model_structure_settings.get(
06         "hidden_units", None)
07     # 默认 Deep 模型的输入为连续型特征+Embedding
08     self.deep_input = self.model_structure_settings.get(
09         "deep_input", "embedding_concat")
10     # 默认 Wide 模型的输入为单纯的连续型特征
11     self.wide_input = self.model_structure_settings.get(
12         "wide_input", "continuous")
13     # 默认 Embedding 的维度 k 为 8
```

```

14     self.embedding_size = self.model_structure_settings.get(
15         "embedding_size", 8)
16
17     # 默认使用 Wide 模型
18     self.use_wide_network = self.model_structure_settings.get(
19         "use_wide_network", self.n_dim > 0)
20     # 由于我们只在 Wide 模型中使用 DNDF (原因会在下一节说明)
21     # 所以如果不使用 Wide 模型的话, 也就不使用 DNDF 了
22     if not self.use_wide_network:
23         self.dndf = None
24     else:
25         # 如果使用 Wide 模型, 就默认也使用 DNDF
26         dndf_params = self.model_structure_settings.get(
27             "dndf_params", {})
28         if self.model_structure_settings.get("use_dndf", True):
29             self.dndf = DNDF(self.n_class, **dndf_params)
30     # 默认使用软剪枝技术
31     if self.model_structure_settings.get("use_pruner", True):
32         pruner_params = self.model_structure_settings.get(
33             "pruner_params", {})
34         self.pruner = Pruner(**pruner_params)
35     # 默认对 DNDF 也应用软剪枝
36     if self.model_structure_settings.get("use_dndf_pruner", True):
37         dndf_pruner_params = self.model_structure_settings.get(
38             "dndf_pruner_params", {})
39         self.dndf_pruner = Pruner(**dndf_pruner_params)
40
41     # 构造 feed dict
42     def get_feed_dict(self, x, y=None, weights=None, is_training=True):
43         # 获取连续型特征
44         # 如果有离散型特征存在的话, 就利用 self.valid_numerical_idx 来将
45         # 所有连续型特征提取出来。注意其最后一位代指标签的类型
46         # 所以要把最后一位排除在外
47         if self._categorical_xs:
48             continuous_x = x[..., self.valid_numerical_idx[:-1]]
49         # 否则, 此时所有特征都是连续型特征, 所以直接赋值即可
50         else:
51             continuous_x = x
52         # 调用 BasicNN 中的方法初始化 feed_dict
53         # 由于 BasicNN 的初始化方法已将 self._is_training 这个对于
54         # Dropout 和 Batch Normalization 而言至关重要的 placeholder
55         # 传进 feed_dict 了, 所以这里我们就无须再传一遍
56         feed_dict = super(Advanced, self)._get_feed_dict(
57             continuous_x, y, weights, is_training)
58         # 如果使用 DNDF 的话, 就要把 self._n_batch_placeholder 这个
59         # 代表 Mini Batch 大小的 placeholder 送进 feed_dict
60         if self._dndf is not None:
61             feed_dict[self._n_batch_placeholder] = len(x)
62         # 将离散型特征送进 feed_dict

```

```

63     for (idx, ), categorical x in zip(
64         self.categorical_columns, self.categorical_xs
65     ):
66         feed_dict.update({categorical x: x[... , idx].astype(np.int32)})
67     return feed_dict

```

如果从前文介绍的技术比较熟悉的话，那么这些代码应该还是比较直观易懂的。

5.5.2 WnD 与 DNDF

在 5.2.4 节的最后，我们总结了 Wide and Deep 模型的两大缺点：

- 需要手动去选取一些交叉特征。
- 容易倾向于训练简单模型（Wide 模型）。

在本节中我们将说明，如果把 WnD 模型与 DNDF 结构相结合的话，那么就能在一定程度上同时解决这两大缺点。

先来看手动选取交叉特征的问题。在 5.3.4 节中我们曾经说过，DNDF 从理论上能包容决策树模型，而交叉特征通常可以用决策树来表达，所以 DNDF 从理论上就能直接表达出交叉特征。这意味着在理想情况下，如果我们让 DNDF 接收的特征层为原始特征的话，DNDF 就能把所有有用的交叉特征都提取出来。换句话说，如果我们把 Wide and Deep 中的 Wide 模型从线性模型换成 DNDF 的话，理想情况下就能自动地选取出交叉特征而无须手动进行挑选。

然后是 Wide and Deep 倾向于训练简单模型的问题。如果我们使用 DNDF 作为 Wide 模型的话，DNDF 的输出是概率输出，而 Deep 模型的输出是普通的输出。换句话说，相比起 Deep 模型的输出而言，DNDF 的输出会小很多。这就意味着此时的 Wide 模型在更多情况下都只是起到一个“辅助决策”的作用（类似于残差的思想），从而不仅 5.2.4 节所提到过的 Wide 模型常常会占据“主导地位”的问题将不复存在，而且“结合浅模型与深模型”这个 WnD 的重要思想，将会进一步升华为“深模型为主，浅模型为辅”的，更符合直观认知的思想。这也是为何我们只对 Wide 模型应用 DNDF，因为如果对 Deep 模型也应用 DNDF 的话，那么 Wide 模型的输出与 Deep 模型的输出的大小就会又回归一致，从而 Wide 模型可能就会又会占据“主导地位”。下面我们就来（不太严谨地）证明这一点。

假设我们现在同时对 Deep 模型和 Wide 模型应用了 DNDF 结构，而且任务是分类问题，再不妨设它们的输出分别为 \mathbf{o}_{deep} 、 \mathbf{o}_{wide} ，那么由于模型的最终输出 \mathbf{o} 通常是 Deep 模型与 Wide 模型输出的加总，所以就有

$$\mathbf{o} = \mathbf{o}_{\text{deep}} + \mathbf{o}_{\text{wide}}$$

注意到此时 \mathbf{o}_{deep} 和 \mathbf{o}_{wide} 都是 DNDF 的输出，即它们都是概率输出，所以一个很自然的想法就是令

$$\mathbf{o} = p \cdot \mathbf{o}_{\text{deep}} + (1 - p) \cdot \mathbf{o}_{\text{wide}}$$

其中， p 满足

$$0 < p < 1$$

它代指 Deep 模型输出的重要性。此时不难看出，模型的最终输出 \mathbf{o} 和 \mathbf{o}_{deep} 与 \mathbf{o}_{wide} 一样，也会是一个概率向量。在此基础上，如果令 p 为可训练的参数的话，由于我们不希望 Wide 模型占据主导地位，所以我们自然期望 p 能够训练得比较大，从而 Deep 模型的输出能够有更大的话语权。不过事实却是，在大多数情况下， p 都会越训练越小，即事实上 Wide 模型会越来越占据主导地位。

具体而言，假设损失函数为

$$L = L(\mathbf{o}, y)$$

那么我们就把 L 对 \mathbf{o} 的梯度求出来，即

$$\nabla_{\mathbf{o}} L$$

将会是一个已知量。那么由

$$\mathbf{o} = p \cdot \mathbf{o}_{\text{deep}} + (1 - p) \cdot \mathbf{o}_{\text{wide}}$$

就可知

$$\begin{aligned} \frac{\partial L}{\partial p} &= \sum_{k=1}^K \frac{\partial L}{\partial o^{(k)}} \cdot \frac{\partial o^{(k)}}{\partial p} \\ &= \sum_{k=1}^K \frac{\partial L}{\partial o^{(k)}} \cdot \frac{\partial}{\partial p} (p \cdot o_{\text{deep}}^{(k)} + (1 - p) \cdot o_{\text{wide}}^{(k)}) \\ &= \sum_{k=1}^K \frac{\partial L}{\partial o^{(k)}} \cdot (o_{\text{deep}}^{(k)} - o_{\text{wide}}^{(k)}) \end{aligned}$$

由于我们要做的是分类问题，所以一个很自然的想法就是使用 log-likelihood 作为损失函数。此时如果假设真实的类别是第 k 类的话，就有

$$L = -\log o^{(k)}$$

从而

$$\frac{\partial L}{\partial p} = \frac{1}{o^{(k)}} \cdot (o_{\text{wide}}^{(k)} - o_{\text{deep}}^{(k)})$$

由于 Wide 模型的模型复杂度比 Deep 模型的复杂度要低，所以 Wide 模型的训练速度会比 Deep 模型的训练速度快。换句话说，在前期的训练过程中，Wide 模型的表现会比 Deep 模型的表现要好，这就意味着

$$o_{\text{wide}}^{(k)} > o_{\text{deep}}^{(k)}$$

从而

$$\frac{\partial L}{\partial p} = \frac{1}{o^{(k)}} \cdot (o_{\text{wide}}^{(k)} - o_{\text{deep}}^{(k)}) > 0$$

此时如果我们使用 Vanilla Update 来更新参数 p 的话，就会有

$$p^{(\text{new})} = p - \eta \cdot \frac{\partial L}{\partial p} < p$$

即 p 将会越训练越小，这就完成了证明。

5.5.3 DNDF 与剪枝

虽然 DNDF 和决策树有着异曲同工之妙，但是它与神经网络的剪枝技术相结合时，对标的并不是决策树中的剪枝算法，而是随机森林中的随机性。具体而言，随机森林在训练每一棵决策树时，所用的特征向量都不是完整的特征向量，而只是特征向量中的一部分。比如，假设完整的特征向量为 $\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(n)})^T$ 的话，那么对于随机森林中的第 t 棵树而言，它所接收的特征可能就是

$$\mathbf{x}_t = (x^{(i_1^{(t)})}, x^{(i_2^{(t)})}, \dots, x^{(i_{n_t}^{(t)})})^T$$

其中， $i_1^{(t)}, \dots, i_{n_t}^{(t)}$ 是 $1 \sim n$ 中的 n_t 个数， n_t 则是一个比 n 小的数，通常的做法是取

$$n_t = \lfloor \log_2 n \rfloor$$

这种接收特征上的随机性，正是随机森林中“随机”二字的来源。而由前文的讨论可知，我们只会在 Wide 模型中用到 DNDF，所以 DNDF 接收的特征层即为原始的特征向量。此时如果对 DNDF 应用剪枝技术的话，节点层所接收的特征也将会是原始特征向量的一部分，因为另外的部分将会被剪枝技术剪掉。换句话说，此时 DNDF 中每棵树所接收的特征也会有随机性。这就意味着 DNDF 不仅自身天生蕴含决策树的表达能力，而且在结合了剪枝技术后还能兼顾随机森林这个集成算法。

为了让大家对 DNDF 的集成能力有一个更直观的认知，我们在此提供一种简单的、具体可行的、应用随机森林来初始化 DNDF 的方法。该做法只是对随机森林算法的一个粗浅应用，由于现实情况非常复杂多样，所以这种做法从实际上来看很有可能不会比原论文所给出的、对 DNDF 中所有 M 棵树都“一视同仁”的方法有根本上的提升。但笔者认为，这种一视同仁多多少少会让这 M 棵树变得比较“同质”。如果能利用随机森林，或是其他集成算法让这 M 棵树变得“异质”的话，最终的效果可能就会比现在的要好得多。

该方法比起原论文一步到位的初始化方法来说，最大的不同在于它会依次初始化 DNDF 中的 M 棵树。具体而言，我们的做法是：

- 先初始化第 1 棵树的参数，然后从总的训练集中有放回地随机抽取一部分数据集作为第 1 棵树的训练集并进行一定的训练（比如，进行 10 次迭代）。
- 固定住第 1 棵树的参数并初始化第 2 棵树的参数，然后只对第 2 棵树进行一定的训练。其中需要注意的是，第 2 棵树的训练集同样是总的训练集中的一个随机采样，它一般会和第 1 棵树用的训练集不一样（这其实就是著名的 Bootstrap Aggregating，被简称为 Bagging 的做法，详情可以参见 https://en.wikipedia.org/wiki/Bootstrap_aggregating）。
- 依此类推，直到完成所有 M 棵树的初始化与预训练。

如果对随机森林算法比较熟悉的话就会发现，如果把上述做法中的主体从 DNDNF 中的树改成决策树的话，那么这一套做法其实基本就是随机森林算法本身，只不过随机森林会让每棵树都进行充分的训练，而由于我们只是想进行初始化工作，所以在上述做法中就没有让每棵树都训练完全，而只是让它们进行了一定的训练。

5.5.4 剪枝与 Dropout

对于比较熟悉 Dropout 但是没有听说过神经网络剪枝技术的人来说，在第一次接触神经网络剪枝时，很有可能会以为它是 Dropout 的一种特殊形式。然而事实上，无论是看理念还是看实际操作，剪枝与 Dropout 都是完全不同的两项技术。具体而言：

- 剪枝的理念是将（全连接）神经网络中的某些不太重要的连接剪掉，从而让模型趋于精简；而 Dropout 的理念是在每个迭代中训练神经网络中的一个“子神经网络”，它的主要目的是让训练算法看到的模型是精简的模型，而不是像剪枝那样把模型本身变得精简。
- 剪枝和 Dropout 在实际操作中都会维护一个权值矩阵的 mask，但是剪枝中的 mask 是权值矩阵的函数，而 Dropout 的 mask 一般都是随机生成的。

总之，我们不能把剪枝与 Dropout 混为一谈。虽然它们之间没有特别好的互补性，但是由于它们对于彼此而言也没有可替代性，所以同时应用这两项技术也是很好的选择。

5.5.5 没有免费的午餐

虽然我们花了非常多的时间来介绍神经网络的各种技术，但是不得不提的一点是：对于任何两个优化算法而言，如果把它们在所有可能的任务上的表现取一个平均的话，就会发现表现是一致的。这就是机器学习界中广为人知的“没有免费的午餐定理”（No free lunch theorem）。换句话说，当我们关心的是所有可能的任务时，就不会有一个模型比另一个模型好。这就意味着不可能有模型能够“吃遍天下”，因为总会有一些模型在某些任务中要表现得更好一些。所以对于 AdvancedNN 而言，虽然它引入了很多很好的技术，理论上也有许多不错的性质，但是 No free lunch theorem 告诉我们，总是会存在这样一个任务，它使得 BasicNN 的表现要比 AdvancedNN 好。

总之在实际应用中，是没有普适的、最好的神经网络结构的，还是要依情况进行分析、设计，不能盲目地相信 AdvancedNN 的有效性。不过，这不代表 AdvancedNN 的发明毫无意义。事实上，正由于它含有诸多优良性质，所以我们虽然不能保证 BasicNN 表现会更好的任务的存在性，但是却可以尽可能地压低这种任务在实际情况中的出现概率，而这也几乎是所有机器学习模型的目标——让自己在实际情况中能以较大的概率比其余模型做得更好。

注意：也正因此，我们会在第 7 章介绍工程化机器学习框架（DistMixIn）时，给出一种朴素但有效的神经网络结构搜索方法。在应用了这种搜索方法后，得到的最终模型一般都会比单纯一股脑地应用 WnD+DNDNF+剪枝的模型要好。

5.6 AdvancedNN 的实际性能

这一节我们会在许多人造数据集上评估同时应用了 WnD、DNDF 与剪枝技术的、增强版神经网络（AdvancedNN）的性能。与第3章类似，虽然我们允许输入数据中有离散型特征，但是还是暂时认为输入连续型特征都是已经经过了数据预处理的特征。

注意：接下来将会用到的核心测试代码可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/e_AdvancedNN/AdvancedNN.ipynb、https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/AdvancedNN/Test.py 与 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/Madelon/TestAdvancedNN.py。

首先要做的，自然就是前文反复用到过的高维噪声线性数据集上的实验了。在 $n = 100$ 、 $k = 5$ 、噪声标准差为 0.5 的数据集下，朴素神经网络（BasicNN）、WnD、WnD+DNDF 和 WnD+DNDF+Surgery 这四种模型的表现将如图 5.22 所示。可以看到，基本上是每新应用一项技术，模型的性能就会提高一些。为了更直观地认知模型训练的过程，我们可以把它们 acc（准确率）曲线画出来，如图 5.23 所示；为了简洁，我们省去了单纯的 WnD 模型的 acc 曲线，因为它基本都比 WnD+DNDF 要差。

BasicNN	acc - Train :	1.0	CV : None	Test : 0.880667
WnD	acc - Train :	0.9885	CV : None	Test : 0.899333
WnD & DNDF	acc - Train :	0.9896	CV : None	Test : 0.901333
WnD & DNDF & Pruner	acc - Train :	0.9512	CV : None	Test : 0.932667

图 5.22 BasicNN与应用了不同新技术的AdvancedNN的最终表现

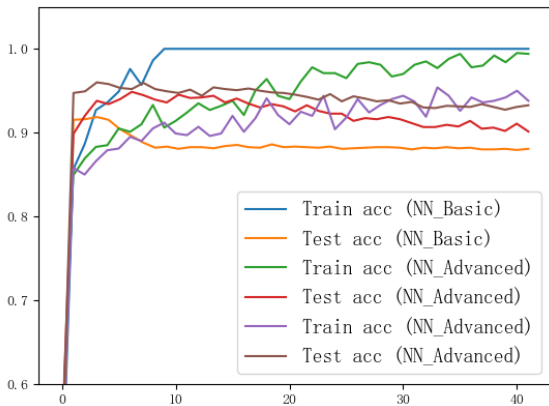


图 5.23 BasicNN与应用了不同新技术的AdvancedNN的acc曲线

注意：acc 曲线和各个模型的对应关系可以结合各个模型的最终表现，并通过 acc 曲线的终点来看出。以图 5.23 为例，由图 5.22 可知，训练集上的模型最终表现的排序为 BasicNN → WnD+DNDF（后简称 DNDF）→ WnD+DNDF+Surgery（后简称 Surgery），所以图 5.23 中终点最高的三个曲线分别对应着它们的训练 acc 曲线；类似的，由于测试集上的性能排序为 Surgery → DNDF → BasicNN，所以终

点后三高的曲线分别对应着它们的测试 acc 曲线。这种对应关系在后文也是通用的，彼时将不再赘述。

可以看到，BasicNN 的训练 acc 会迅速达到 100%，但是从测试 acc 可以看出，其过拟合现象也非常严重。反观 DNDF 与 Surgery，虽然它们的训练 acc 上升得并不快，但是它们的测试 acc 下降得也比较慢，这说明它们对噪声的耐受力要比 BasicNN 强很多。而至于 DNDF 与 Surgery 之间的对比的话，可以看出，Surgery 会比 DNDF 要好。这恰恰从实验的角度佐证了 5.4.4 节最后提供的理论证明——即剪枝技术能够把真正重要的权值保留下来，而把其余的冗余权值给剪掉。同时，由于剪枝技术能让模型趋于精简，所以它的泛化能力自然也就更强。

为了更全面地说明这一点，我们把特征总维度提升至 $n = 500$ ，那么此时就只有 1%是有用的特征，即有 99%的特征都是冗余的特征。在这种情况下，BasicNN、DNDF 和 Surgery 的性能对比与 acc 曲线将会如图 5.24 和图 5.25 所示（为突出重点，我们没有画出训练 acc 曲线）。可以看到，虽然它们都过拟合了，但是加了剪枝之后的泛化能力还是明显比没加剪枝时要高出一截。

BasicNN	acc - Train :	1.0	CV : None	Test :	0.9
WnD & DNDF	acc - Train :	1.0	CV : None	Test :	0.904667
WnD & DNDF & Pruner	acc - Train :	0.9864	CV : None	Test :	0.928667

图5.24 各模型在高冗余特征下的最终表现

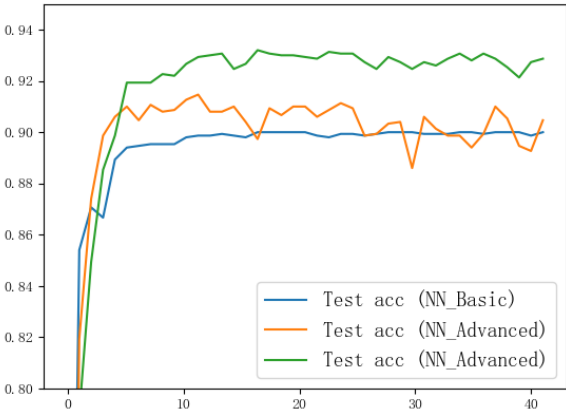


图5.25 各模型在高冗余特征下的acc曲线

而如果我们在此基础上把噪声的标准差调整成 1 的话，加入 Surgery 的效果就会变得更加明显一些（如图 5.26 和图 5.27 所示），因为此时不仅有 99%的冗余特征，而且剩下的 1%的特征还都带着非常大的噪声，因此 Surgery 的两大优势——挑选特征与防止过拟合就会体现得淋漓尽致。

BasicNN	acc - Train :	1.0	CV : None	Test :	0.819333
WnD & DNDF	acc - Train :	1.0	CV : None	Test :	0.81
WnD & DNDF & Pruner	acc - Train :	0.9641	CV : None	Test :	0.85

图5.26 各模型在高冗余、高噪声特征下的最终表现

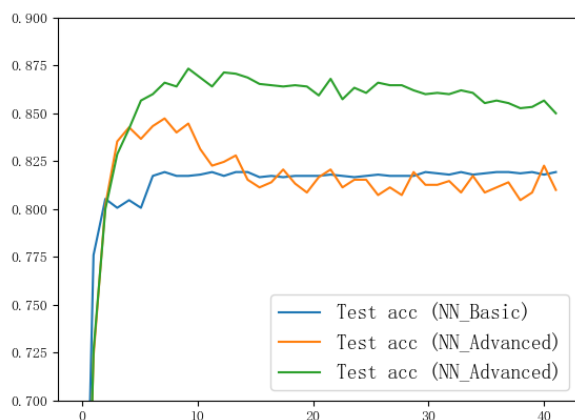


图5.27 各模型在高冗余、高噪声特征下的acc曲线

不过反过来说，如果我们不加进冗余特征（即 $k = n = 100$ ），且将噪声的标准差设置为 0（即不加入噪声）的话，那么 Surgery 的效果可能反而会差一些（如图 5.28 和图 5.29 所示），因为此时 Surgery 会把真正有用的特征所对应的权值连接给剪掉。而如果我们把 Surgery 换成软剪枝技术的话，效果相比之下就会好一些（如图 5.30 和图 5.31 所示）。这其实就佐证了我们前文阐述过的一个观点：Surgery 在特征已经做得较好的数据集上会出现性能损失，而软剪枝则缓解了这个问题。

BasicNN	acc - Train :	1.0	CV : None	Test : 0.970667
WnD & DNDF	acc - Train :	0.9999	CV : None	Test : 0.974
WnD & DNDF & Pruner	acc - Train :	1.0	CV : None	Test : 0.968

图5.28 各模型在无冗余特征下的最终表现（使用Surgery技术）

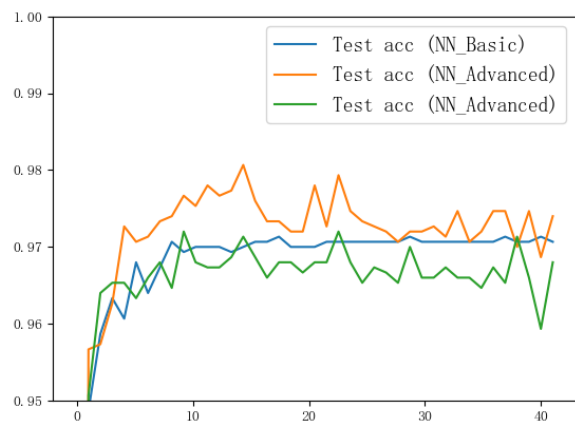


图5.29 各模型在无冗余特征下的acc曲线（使用Surgery技术）

BasicNN	acc - Train :	1.0	CV : None	Test : 0.962667
WnD & DNDF	acc - Train :	1.0	CV : None	Test : 0.964667
WnD & DNDF & Pruner	acc - Train :	1.0	CV : None	Test : 0.969333

图5.30 各模型在无冗余特征下的最终表现（使用软剪枝技术）

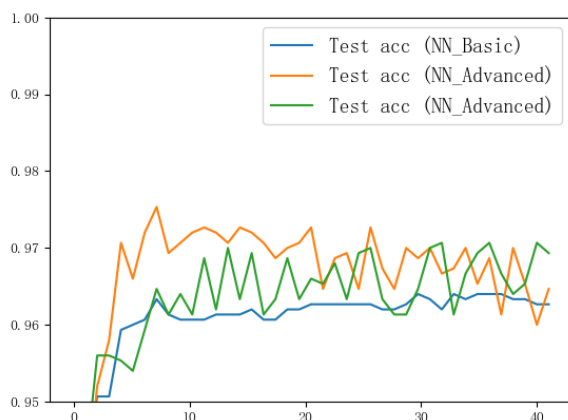


图5.31 各模型在无冗余特征下的acc曲线（使用软剪枝技术）

不过由于高维噪声线性数据集的内在规律仍然是线性的规律，所以大体上来说这个数据集还是比较简单的；为了测试更难的数据集上 AdvancedNN 的性能，我们可以类似地定义一个“高维噪声多项式数据集”，其生成规则为：

- 使用一维标准正态分布来采样出原始的 n 维特征向量 \mathbf{x} 。
- 在 \mathbf{x} 的基础上，加上服从均值为 0、标准差为 0.5 的正态分布噪声，得到训练所用的特征向量 \mathbf{x} 。
- 在 n 个维度 $(x^{(1)}, x^{(2)}, \dots, x^{(n)})$ 中挑选出 k 个 $(x^{(i_1)}, x^{(i_2)}, \dots, x^{(i_k)})$ 作为真正有用的特征：

$$\hat{\mathbf{x}} \triangleq (x^{(i_1)}, x^{(i_2)}, \dots, x^{(i_k)})^T$$

- 将这 k 个真正有用的特征值的平方、立方、4 次方……一直到 p 次方的值算出来：

$$\hat{\mathbf{x}}^j \triangleq \hat{\mathbf{x}}^j = \left((x^{(i_1)})^j, (x^{(i_2)})^j, \dots, (x^{(i_k)})^j \right)^T, \quad j = 1, 2, \dots, p$$

- 随机初始化 p 个权值向量

$$\mathbf{w}_j = (w_j^{(1)}, w_j^{(2)}, \dots, w_j^{(k)})^T, \quad j = 1, 2, \dots, p$$

并根据

$$\sum_{j=1}^p \mathbf{w}_j^T \hat{\mathbf{x}}_j$$

是否大于 0 来给相应的带噪声的特征向量 $\hat{\mathbf{x}}$ 打上标签 y ：

$$y = \begin{cases} 1 & , \text{ if } \sum_{j=1}^p \mathbf{w}_j^T \hat{\mathbf{x}}_j > 0 \\ 0 & , \text{ if } \sum_{j=1}^p \mathbf{w}_j^T \hat{\mathbf{x}}_j \leq 0 \end{cases}$$

这就得到了我们的训练集：

$$D = \{(\tilde{x}_1, y_1), (\tilde{x}_2, y_2), \dots, (\tilde{x}_N, y_N)\}$$

- 至于测试集，则是不带噪声的特征向量与标签的组合。

不难发现，高维噪声线性数据集其实就是 $p = 1$ 时的高维噪声多项式数据集。当我们把 p 调高时，对应的高维噪声多项式数据集的难度也会随之变高。先来看看 $p = 3$ 时的结果（如图 5.32 和图 5.33 所示；为方便比较，我们仍取 $n = 100$ 、 $k = 5$ ，剪枝技术也仍然用 Surgery），可以看到即使是在比较难的数据集上，DNDF 与 Surgery 这两个 AdvancedNN 也会比朴素的 BasicNN 的性能要好不少。

BasicNN	acc - Train :	1.0	CV : None	Test : 0.804667
WnD & DNDF	acc - Train :	0.9781	CV : None	Test : 0.834
WnD & DNDF & Pruner	acc - Train :	0.9309	CV : None	Test : 0.849333

图 5.32 各模型在3次多项式数据集下的最终表现

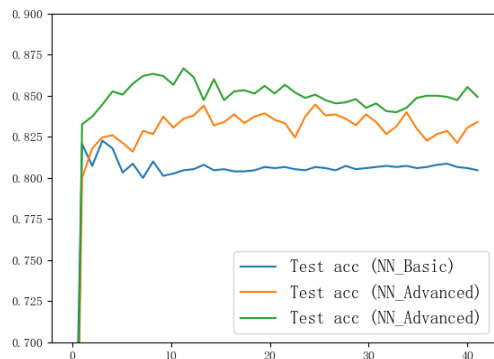


图 5.33 各模型在3次多项式数据集下的acc曲线

而即使我们不断地加大难度，AdvancedNN 也能稳定地比 BasicNN 要好，而且好的程度有越来越大的趋势。比如，如果我们依次对 $p = 5$ 、 $p = 8$ 、 $p = 12$ 做实验的话，结果将如图 5.34~图 5.39 所示。

BasicNN	acc - Train :	1.0	CV : None	Test : 0.730667
WnD & DNDF	acc - Train :	0.9732	CV : None	Test : 0.748667
WnD & DNDF & Pruner	acc - Train :	0.9091	CV : None	Test : 0.793333

图 5.34 各模型在5次多项式数据集下的最终表现

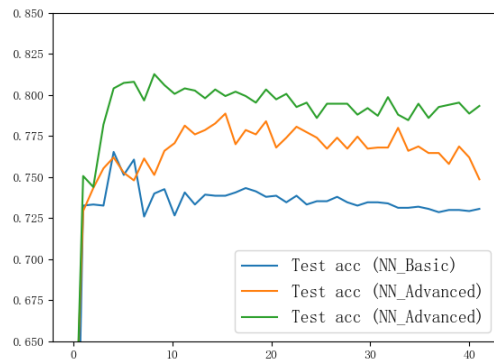


图 5.35 各模型在5次多项式数据集下的acc曲线

BasicNN	acc - Train :	1.0	CV : None	Test :	0.696
WnD & DNDF	acc - Train :	0.9649	CV : None	Test :	0.726
WnD & DNDF & Pruner	acc - Train :	0.8907	CV : None	Test :	0.762667

图5.36 各模型在8次多项式数据集下的最终表现

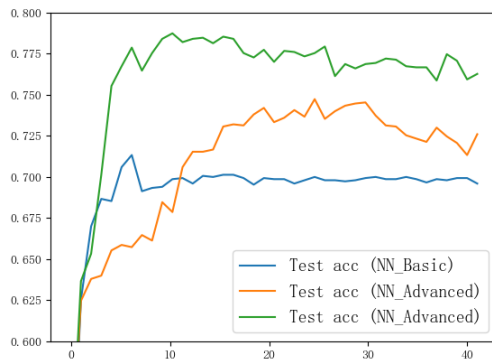


图5.37 各模型在8次多项式数据集下的acc曲线

BasicNN	acc - Train :	1.0	CV : None	Test :	0.670667
WnD & DNDF	acc - Train :	0.9586	CV : None	Test :	0.681333
WnD & DNDF & Pruner	acc - Train :	0.8852	CV : None	Test :	0.725333

图5.38 各模型在12次多项式数据集下的最终表现

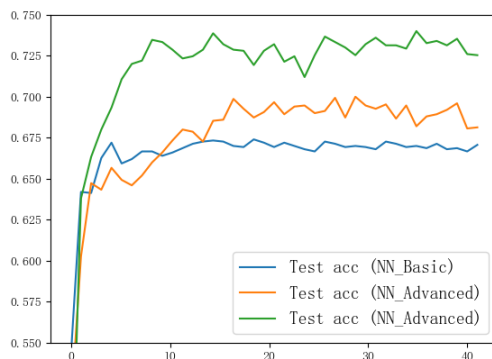


图5.39 各模型在12次多项式数据集下的acc曲线

综上所述，我们就能够比较有自信地说：**AdvancedNN** 比起 **BasicNN** 而言，可以（即使在有噪声的干扰下）更好地学习出多项式系列的规律。

至于其他情形下 **AdvancedNN** 的实际性能，前面曾屡次用过的 **Madelon** 数据集将会是一个比较好的衡量标准。由前文的讨论可知，**BasicNN** 在 **Madelon** 数据集原始特征向量下的表现为

$$\text{acc}_{\text{tr}} \approx 55\%, \quad \text{acc}_{\text{te}} \approx 53\%$$

在对原始特征向量做了标准化处理后，其表现为

$$\text{acc}_{\text{tr}} = 100\%, \quad \text{acc}_{\text{te}} \approx 56.3\%$$

即出现了严重的过拟合现象。注意，我们本章第一个叙述的技术——**Dropout** 就是用于防止过拟合的，所以一个很自然的想法就是用引入了 **Dropout** 的 **AdvancedNN** 来做实验。当我们把

Dropout 中的 `keep_prob` 设置为默认值 0.5 时, AdvancedNN 与朴素的 BasicNN 的最终表现与 acc 曲线将如图 5.40 和图 5.41 所示。

BasicNN	acc - Train :	1.0	CV : None	Test :	0.565
AdvancedNN	acc - Train :	1.0	CV : None	Test :	0.578333

图5.40 AdvancedNN (`keep_prob=0.5`) 与 BasicNN 在 Madelon 数据集上的最终表现

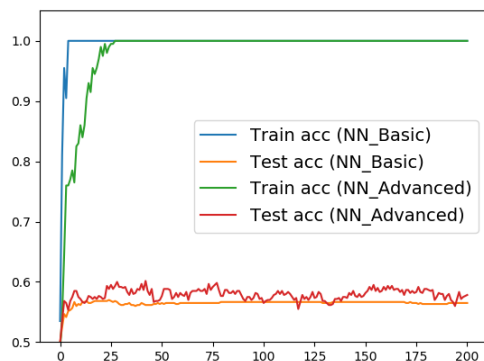


图5.41 AdvancedNN (`keep_prob=0.5`) 与 BasicNN 在 Madelon 数据集上的 acc 曲线

可以看到, 加了 Dropout 的 AdvancedNN 在训练集上的收敛速度会慢一些, 但是在测试集上的表现会普遍比 BasicNN 要好 ($\text{acc}_{\text{te}} \approx 58.0\%$)。不过不难看出, 模型的过拟合现象仍然非常严重, 所以一个自然的想法就是把 `keep_prob` 降低, 从而期望 Dropout 能带来更好的泛化能力。事实上, 如果我们将 `keep_prob` 降到 0.25 的话, 结果将如图 5.42 和图 5.43 所示。可以看到, 此时 AdvancedNN 的训练收敛速度会更慢, 但测试集上的表现会更好 ($\text{acc}_{\text{te}} \approx 58.6\%$), 这是符合 Dropout 的直观性质的。

BasicNN	acc - Train :	1.0	CV : None	Test :	0.565
AdvancedNN	acc - Train :	1.0	CV : None	Test :	0.581667

图5.42 AdvancedNN (`keep_prob=0.25`) 与 BasicNN 在 Madelon 数据集上的最终表现

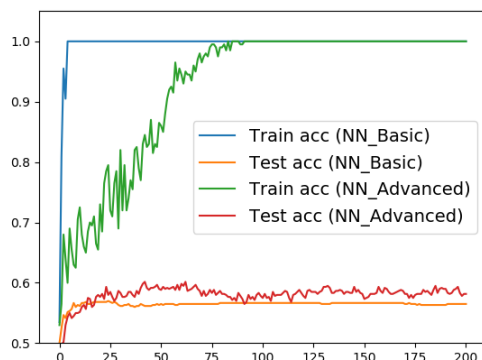


图5.43 AdvancedNN (`keep_prob=0.25`) 与 BasicNN 在 Madelon 数据集上的 acc 曲线

但是, 并不是说 `keep_prob` 越小越好。事实上, 如果我们将 `keep_prob` 设置得太小, 就会产生“过犹不及”的后果。具体而言, 如果把 `keep_prob` 降到 0.1 的话, 那么结果就将如图 5.44

和图 5.45 所示。可以看到，此时 AdvancedNN 的训练收敛速度会比 BasicNN 慢几十倍，而测试集上的表现却差不多 ($\text{acc}_{\text{te}} \approx 56.4\%$)。

BasicNN	acc - Train :	1.0	CV : None	Test :	0.565
AdvancedNN	acc - Train :	0.9995	CV : None	Test :	0.563333

图5.44 AdvancedNN (keep_prob=0.1) 与 BasicNN 在 Madelon 数据集上的最终表现

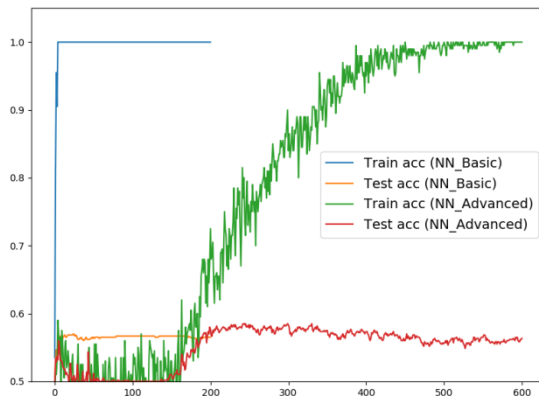


图5.45 AdvancedNN (keep_prob=0.1) 与 BasicNN 在 Madelon 数据集上的 acc 曲线

综上所述我们可以大致得出这样的结论：单单引入 Dropout 这一项技术，只能把性能提升到 $\text{acc}_{\text{te}} \approx 58.6\%$ ，此时 keep_prob 为 0.25。由于我们知道 Madelon 数据集的原始特征向量非常不适合作为神经网络的输入，而必须先做标准化，那么从直观上来说，如果在此时再引入 Batch Normalization 的话，那么神经网络的表现可能就会更上一层楼，而事实上也确实如此，如图 5.46 和图 5.47 所示。可以看到，此时不仅训练收敛速度稍微快了一点，测试集上的准确率也突破了 60 大关 ($\text{acc}_{\text{te}} \approx 60.3\%$)。

BasicNN	acc - Train :	1.0	CV : None	Test :	0.565
AdvancedNN	acc - Train :	1.0	CV : None	Test :	0.603333

图5.46 AdvancedNN (keep_prob=0.25+BN) 与 BasicNN 在 Madelon 数据集上的最终表现

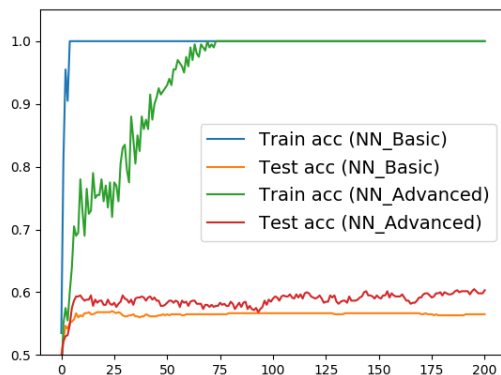


图5.47 AdvancedNN (keep_prob=0.25+BN) 与 BasicNN 在 Madelon 数据集上的 acc 曲线

接下来要做的，自然就是同时应用上 Wide and Deep、DNDF 和剪枝，此时 AdvancedNN 的表

现将如图 5.48 和图 5.49 所示。

BasicNN	acc - Train :	1.0	CV : None	Test : 0.563333
AdvancedNN	acc - Train :	1.0	CV : None	Test : 0.618333

图5.48 AdvancedNN (keep_prob=0.25+BN+WnD+DNDF+Pruning)
与BasicNN在Madelon数据集上的最终表现

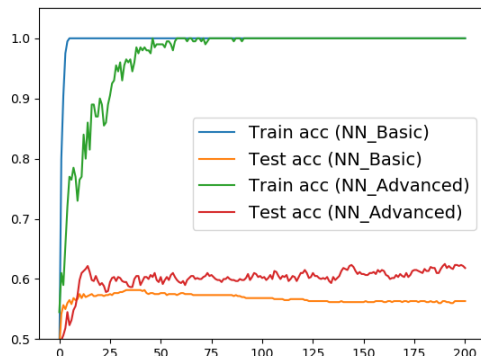


图5.49 AdvancedNN (keep_prob=0.25+BN+WnD+DNDF+Pruning)
与BasicNN在Madelon数据集上的acc曲线

可以看到，此时训练收敛速度又快了一点，且测试集上的表现也好了不少 ($\text{acc}_{\text{te}} \approx 62\%$)。值得一提的是，从图 5.49 中还可以看出，此时的 AdvancedNN 在测试集上的表现还有继续上升的趋势，这较好地佐证了软剪枝技术确实能增强模型的泛化能力。

5.7 本章小结

- 朴素神经网络 (BasicNN) 存在如下根本性的问题：
 - 表达能力太强以至于容易陷入过拟合。
 - 反向传播中梯度过于敏感。
 - 网络模型的结构过于单一。
 - 层与层之间是全连接的连接形式。
- Dropout 能在一定程度上防止神经网络陷入过拟合，Batch Normalization 能使神经网络中每一层接收到的输入都比较“合适”。这两者解决了前两个根本性问题。
- Wide and Deep 的重要思想是结合“浅”模型与“深”模型，从而使神经网络能够适应于不同难度的数据集。
- DNDF 能够包容决策树的表达能力，而且有结合集成算法的可能性，能够加以应用的场景非常广泛。WnD 和 DNDF 都解决了第三个根本性问题。
- 神经网络中的剪枝技术能够在一定程度上剪掉冗余的权值，并把真正重要的权值保留下来，它能使神经网络趋于精简，从而提高模型的泛化能力。

- Surgery 的初衷在于压缩模型而非提升模型性能，所以我们设计了专门用于提升模型性能的剪枝技术——软剪枝，它拥有如下性质：
 - 比起 Surgery，它在诸多方面都引入了“过渡”这个重要的思想。这种“过渡”使得软剪枝下的权值可能大多集中在剪枝阈值的附近“待命”，从而使得整个剪枝过程更加动态、更加符合直觉。
 - 通过调整超参数（ $\alpha \rightarrow 0$ 、 $\beta \rightarrow \infty$ ），它能够比较好地逼近 Surgery 的表现。
 - 解决了第四个根本性问题。
- 各项技术并不完全是独立存在的，它们彼此之间可能会有较好的互补性。将它们放在一起的话，就能得到适用范围非常广的 AdvancedNN。具体而言：
 - WnD 需要手动选取交叉特征，且浅模型可能会“喧宾夺主”，此时如果把 Wide 模型从 Logistic 回归换成 DNDF 的话，就能比较好地缓解这两个缺点的程度，因为 DNDF 既能在一定程度上自动挑选特征，又能让模型的思想“升华”为“深模型为主，浅模型为辅”这样一个更符合直观认知的思想。
 - DNDF 加上剪枝技术，能在一定程度上包容随机森林这个强大的集成算法。
 - 剪枝技术与 Dropout 有相似之处但本质不同，它们之间虽然起不到“ $1 + 1 > 2$ ”的效果，但至少能保证“ $1 + 1 \approx 2$ ”。

第 6 章

半自动化机器学习框架

在此之前的章节中，我们介绍了许多机器学习的理论与技术，而对于其中的神经网络部分，更是花了比较大的力气去叙述其方方面面。不过，虽然它们确实功能相当强大，适用范围也很广，但是应用起来却还是有诸多不便。比如在使用 `AdvancedNN` 时，我们不仅仍然需要先对数据做很多预处理工作，而且还需要告诉 `AdvancedNN` 哪些是离散型特征、哪些是连续型特征。对于一个理想的机器学习框架来说，我们自然希望机器能够自动地帮助我们处理这一整套准备工作，这也正是本章所要完成的任务。

具体而言，本章所要介绍的半自动化机器学习框架（`AutoBase`）旨在配合第 3 章实现的 `TensorFlow` 模型的基本框架（`Base`）来使用，它能够在不对基于 `Base` 拓展的模型（比如第 3 章和第 5 章介绍的 `BasicNN` 和 `AdvancedNN`，以及附录 A 中将会介绍的 `SVM` 系列的模型）的代码做出任何改变的同时，将数据预处理的整套逻辑嵌入该模型中。换句话说，只要是基于 `Base` 拓展的模型，那么利用 `AutoBase`，我们就能把它“升级”为半自动化的模型。为方便叙述，我们后文会统一使用单词“`model`”代指基于 `Base` 拓展的某个机器学习模型（比如 `AdvancedNN`），于是 `AutoBase` 的目的就是让 `model` “升级”为半自动化模型。

那么究竟何谓半自动化模型呢？所谓的半自动化模型，在这里指的主要是 `model` 接收的输入可以是原始的、文件格式的数据集，而不一定是处理好的、`Python` 或 `numpy` 格式的数据。然后，半自动化的 `model` 就可以在其内部自动完成所有数据预处理工作，而且这个处理过程可以很好地在测试集上复现。而之所以称其为“半自动化”而非“自动化”，是因为它并没有包含“自动挑选、优化模型超参数”的过程。相对应的，其自动化主要体现在“自动数据预处理”上。

此外对于许多 `model` 而言，训练的算法都是迭代算法（比如梯度下降算法）。迭代算法虽然有不少优点，但是它也会带来一个很大的痛点——那就是迭代次数的选择。作为一个足够好的半自动化框架，`AutoBase` 需要能够帮助我们自动选取迭代次数，从而能让我们专注于 `model` 本身的调优上。幸运的是，我们在 3.5.3 节的代码 3.3 中用到的 `TrainMonitor` 就已经能够做到“自

动选取迭代次数”。具体而言，它能够利用 `model` 的历史表现来决定当前是应该停止迭代还是继续迭代甚至延长迭代次数。不过在第 3 章里我们没有对其实现进行具体说明，所以在本章中我们会进行相应的补充。

需要指出的是，本章（以及第 7 章）的内容均属于工程应用类型的内容而非算法层面上的内容，所以代码及相应实现思想的展示会占大部分，而神经网络本身的说明则几乎没有。不过也正因此，本章及第 7 章所介绍的技术都是非常通用的技术。这些技术的应用场景非常广泛，它们不仅能作用于神经网络模型上，也能作用于几乎任意一个机器学习模型上。完整的代码可以直接参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/Base.py 中的 `AutoBase` 与 `AutoMeta` 部分，笔者也建议大家把完整的代码下载下来以对它有一个总体上的把握。

本章主要涉及的知识点：

- 原始数据的读取与准备
- 非数值型特征的数值化与冗余特征的去除
- 缺失值处理与连续型特征的预处理
- 不平衡数据与稀疏数据的处理
- 元类（Meta Class）的应用
- 训练过程的“监控”

6.1 数据的准备

无论是做什么数据预处理工作，第一步要做的都是把原材料——数据本身准备好。所以在 6.1 节和 6.2 节中，我们会对如何把数据准备成想要的形式做较为详细的说明。具体而言，我们会先在 6.1.1 节和 6.1.2 节概述整个数据预处理和整个数据准备的流程，然后在下一节里补充说明本节没有展开叙述的、和数据转换相关的部分。

6.1.1 数据预处理的流程

为了让大家对本章的内容有一个大致的认知，我们会先在本节里从整体上来说明 `AutoBase` 的逻辑。由于此时注重的是对整个流程的把握，所以我们在说明的过程中可能会跳过许多细节。在后续的章节中，我们将会展开说明这些本节暂时略过的部分。

此前我们在做各种实验时，往往都会假设“所有数据都已经做好数据预处理”；而在这一章中，我们的主要目的就是自动化这个预处理的过程。所以不难想象的是，`AutoBase` 需要实际用到数据的方法中先准备好相应的数据，然后再调用 `model` 中相应的方法。比如，对于 `self.init_from_data` 这个十分关键的、利用数据来初始化相应属性的方法而言，我们就需要先把数据准备好，然后再调用 `model` 的初始化方法：

```
01     def init_from_data(self, x, y, x_test, y_test, sample_weights, names):  
02         # 初始化数据超参数
```

```

03     self.init_data_info()
04     # 调用 self._auto_init_from_data 方法, 完成数据的准备
05     x, y, x_test, y_test = self._auto_init_from_data(
06         x, y, x_test, y_test, names)
07     # 调用 model 的相应方法, 完成相应属性的初始化
08     model.init_from_data(
09         self, x, y, x_test, y_test, sample_weights, names)

```

其中, `self._auto_init_from_data` 是 `AutoBase` 的核心方法之一, 它整合了另外 3 个核心方法, 并能得到前文一直假设可以得到的“经过了数据预处理的数据”。为了方便大家理解, 我们先来对该方法的实现逻辑进行整理, 然后再展示具体的代码。

- 第 1 步是调用第 1 个核心方法 (`self._load_data`) 并做好数据的准备。具体而言, 如果用户没有显式提供 Python 或 numpy 格式的数据的话, 那么就需要将相应文件格式的数据转换为准备好的数据, 此时会把 `None` 传入 `self._load_data`, 并一般会根据某个参数自动地划分出交叉验证集。而如果用户显式地提供了 Python 或 numpy 格式的数据, 则需要分两种情况讨论:
 - 一是用户只提供了训练集, 那么此时就直接把训练集传入 `self._load_data`。
 - 二是用户同时提供了训练集和交叉验证集, 那么此时就需要把训练集和交叉验证集“打包”成一个元组 (tuple) 并将其传入 `self._load_data`。这样一来, 只要 `self._load_data` 看到的是元组, 那么就意味着用户提供了交叉验证集。
- 第 2 步、第 3 步是调用第 2 个、第 3 个核心方法来处理不平衡数据与稀疏数据。

在知道了实现思路后, 相应的代码就比较直观了:

```

01     """
02     实现数据准备的自动化
03     x、y: 训练所用的特征向量与标签
04         由于我们接收的输入可以是文件格式的数据集
05         所以这两者可以为 None
06     x_test、y_test: 交叉验证所用的特征向量与标签
07     names: 训练集与交叉验证集的名字, 后文会给出相应的解释
08     此外, 方法中出现的各个变量的具体意义如下。
09     stage: 代指数据预处理所处的“阶段”, 后文会给出相应的解释
10     shuffle: 代指是否打乱训练数据集
11     file_type: 代指文件格式的数据集的后缀, 默认为 txt
12     test_rate: 代指从训练集划分出交叉验证集时, 交叉验证集所占的比例
13     """
14     def _auto_init_from_data(self, x, y, x_test, y_test, names):
15         stage = self.data_info["stage"]
16         shuffle = self.data_info["shuffle"]
17         file_type = self.data_info["file type"]
18         test_rate = self.data_info["test rate"]
19         # 准备好核心方法 self._load_data 所要用到的参数
20         args = (self.numerical_idx, file_type, names, shuffle, test_rate, stage)
21         # 调用 self._load_data 来获得准备好的数据

```

```

22     if x is None or y is None:
23         # 如果用户没有提供训练集的话,
24         # 就将 None 作为 self. load data 的第 1 个参数传入
25         x, y, x_test, y_test = self. load data(None, *args)
26     else:
27         # 否则, 把 x 和 y 合并成 data
28         data = np.hstack([x, y.reshape([-1, 1])])
29         # 并且在提供了交叉验证集时, 把 data 赋值为一个元组 (tuple)
30         # 该元组的第一个元素为训练集, 第二个元素为交叉验证集
31         if x_test is not None and y_test is not None:
32             data = (data, np.hstack([x_test, y_test.reshape([-1, 1])]))
33         # 把 data 作为 self. load data 的第 1 个参数传入
34         x, y, x_test, y_test = self. load data(data, *args)
35     # 处理不平衡数据
36     self. handle unbalance(y)
37     # 处理稀疏数据
38     self. handle sparsity()
39     return x, y, x_test, y_test

```

数据准备的关键几乎全在 `self.load_data` 这一个方法上, 我们会在下一节中介绍该方法的实现。

6.1.2 数据准备的流程

由上一节的讨论可知, 数据准备的核心方法——`self.load_data` 接收的第 1 个参数有如下 3 种可能的取值类型。

- `None`: 意味着我们要读取文件格式的数据。
- `Python` 或 `numpy` 数组: 意味着用户提供了训练集但没提供交叉验证集。
- 元组 (`tuple`): 意味着用户既提供了训练集, 又提供了交叉验证集, 且训练集和交叉验证集分别是该元组的第 1 个元素和第 2 个元素。

由于从文件中读取数据是一件比较耗时的事情, 再加上后续的许多数据准备工作都需要比较庞大的计算力, 所以一个很自然的想法就是: 把能够重复利用的处理结果缓存在某个地方, 这样如果下次准备数据时满足复用条件的话, 我们就能直接读取这些缓存, 而不用从头开始准备数据。

基于这种缓存的思想, `AutoBase` 中 `self.load_data` 的前期工作就比较直观了。

```

01     """
02     数据准备的核心方法
03     data: 数据集, 其三种可能的取值类型已在上文给出
04     numerical_idx: 意义与 self.numerical_idx 一致, 只不过由于
05         我们的目的是做到数据准备的自动化, 所以它可以是 None
06     file_type: 代指文件格式的数据集的后缀, 默认为 txt
07     names: 训练集与交叉验证集的名字
08     shuffle: 代指是否打乱训练数据集
09     test_rate: 代指从训练集划分出交叉验证集时, 交叉验证集所占的比例

```

```

10     stage: 代指数据预处理所处的“阶段”
11     """
12     def load_data(self, data=None, numerical_idx=None,
13                   file_type="txt", names=("train", "test"), shuffle=True,
14                   test_rate=0.1, stage=3
15     ):
16         # 用一个属性来记录是否利用了缓存
17         use_cached_data = False
18         # 定义两个属性来储存训练集与交叉验证集
19         train_data = test_data = None
20         # 定义各种缓存的路径
21         # 这里的 self.data_folder 为文件格式的数据所处的路径
22         data_cache_folder = os.path.join(self.data_folder, "Cache", self.name)
23         data_info_folder = os.path.join(self.data_folder, "DataInfo")
24         # 定义各种缓存对应的文件名
25         data_info_file = os.path.join(data_info_folder, "{}.info".format(self.name))
26         train_data_file = os.path.join(data_cache_folder, "train.npy")
27         test_data_file = os.path.join(data_cache_folder, "test.npy")

```

以上我们就准备好了缓存的路径与缓存的文件名，从而在后续的实现中，我们就能根据它们与相应的检查来判断是否可以读取缓存了。此外需要特别指出的是，上述代码中用到的 `stage` 变量是数据预处理的核心控制变量，它能帮助我们处理处于不同状态下的数据。为了方便大家理解后续的实现，对它各个取值下的具体意义都进行一个简介是有必要的。

- **stage = 1**（第 1 阶段）：此时接收的数据是新的数据，而且我们关注的重点是从数据中提取出相应的信息，并把这些信息进行相应的缓存。在提取完信息之后，我们会（利用 `self.transform_data` 方法）进行非数值型离散型特征的数值化，以及冗余特征的去除。
- **stage = 2**（第 2 阶段）：此时接收的数据是旧的数据，而且相应的数据信息和一部分数据转换也已经（由第 1 阶段）提取并处理完毕，所以我们不会从缓存文件中读取数据信息，并只把关注的重点放在后续的缺失值处理与连续型特征的预处理上。
- **stage = 3**（第 3 阶段）：等价于连续进行第 1 阶段与第 2 阶段。具体而言，此时接收的数据既可能是新数据，也可能是旧数据。如果没有相应的缓存，说明此时接收的是新数据，所以就进入第 1 阶段来从头完成数据的准备；反之，如果相应的缓存已经存在，说明此时接收的数据是旧数据，所以就进入第 2 阶段。此外，第 3 阶段会同时进行第 1 阶段和第 2 阶段所做的两步数据转换，即它会同时完成：
 - 非数值型离散型特征的数值化与冗余特征的去除。
 - 缺失值的处理与连续型特征的预处理。

注意：这种对数据预处理过程分阶段的思想在工程化应用时非常重要。事实上在下一章介绍工程化机器学习框架（DistMixIn）时，我们会反复利用这种分阶段的数据预处理来完成重复实验的管理。

总之，第 1 阶段意味着从头开始直到准备好“数值型的、无冗余的数据”（后文统一简称它为“原始数据”），第 2 阶段意味着已有原始数据与相应的数据信息，并会在此基础上进行缺失

值处理与数据预处理，而第 3 阶段则是这两者的糅合。

在知道了各个阶段的内涵之后，相应的代码实现就会好理解一些了。

```
28     # 如果用户没有提供数据集，而且处于第 2 阶段或第 3 阶段的话
29     # 那么只要缓存文件存在，就读取相应的缓存文件
30     if data is None and stage >= 2 and os.path.isfile(train_data_file):
31         print("Restoring data")
32         use_cached_data = True
33         train_data = np.load(train_data_file)
34         if not os.path.isfile(test_data_file):
35             test_data = None
36             data = train_data
37         else:
38             test_data = np.load(test_data_file)
39             data = (train_data, test_data)
40     # 如果没能读取到缓存文件，就需要进行进一步处理
41     if not use_cached_data:
42         # 如果没有提供数据集的话
43         # 就调用 self.get_data_from_file 方法来从文件中读取数据
44         if data is None:
45             # 此时读取的数据将会是 Python 的 list 而非 numpy 数组
46             is_ndarray = False
47             data, test_rate = self.get_data_from_file(
48                 file_type, test_rate)
49         else:
50             # 否则，如果 data 不是 tuple 的话
51             # 直接把它转换成 numpy 数组
52             if not isinstance(data, tuple):
53                 # 此外，既然此时用户只提供了训练集，那么认为
54                 # 此时用户只想进行训练，不想进行交叉验证是合理的
55                 # 所以，应该把 test_rate 置为 0
56                 test_rate = 0
57                 data = np.asarray(data, dtype=np.float32)
58             # 而如果 data 是 tuple 的话
59             # 就把它的所有元素都转换成 numpy 数组
60             else:
61                 data = tuple(
62                     arr if isinstance(arr, list) else
63                     np.asarray(arr, np.float32) for arr in data
64                 )
65             # 总之，此时得到的 data 的构成都将会是 numpy 数组
66             is_ndarray = True
67     # 如果 data 是 tuple
68     # 就意味着用户同时提供了训练集和交叉验证集
69     if isinstance(data, tuple):
70         # 此时如果要打乱，就只用打乱其第 1 个元素（训练集）
71         # 注意要根据 data 是否是 numpy 数组来改变行为
72         if shuffle:
73             if is_ndarray:
```



```

74         np.random.shuffle(data[0])
75     else:
76         random.shuffle(data[0])
77     # 用变量 n_train 来存储训练集的样本个数
78     n_train = len(data[0])
79     # 然后, 把训练集和交叉验证集合并成一个总的数据集
80     # 注意要根据 data 是否是 numpy 数组来改变行为
81     if isinstance(data, np.ndarray):
82         data = np.vstack(data)
83     else:
84         data = data[0] + data[1]
85     # 否则, 就意味着只有训练集
86     else:
87         if shuffle:
88             if isinstance(data, np.ndarray):
89                 np.random.shuffle(data)
90             else:
91                 random.shuffle(data)
92         # 此时就根据 test_rate 的设置算出训练集的样本数
93         # 而如果不使用交叉验证集(即 test_rate 为 0)的话
94         # 就把 n_train 设置为-1
95         n_train = int(len(data) * (1 - test_rate)) if test_rate > 0 else -1

```

在上面这段代码中, 我们用到了从 `self.get_data_from_file` 这个从文件中获取数据的方法。如果只是说其核心部分的话, 其实 3.5.2 节里实现的 Toolbox 中的 `get_data` 方法就已经满足基本需求了; 不过既然目标是实现半自动化的框架 (AutoBase), 我们需要支持的功能自然要更加丰富一些。具体而言, 现在已知装数据的文件夹的名字叫 `self.data_folder` (后简称 `sdf`), 且我们的目标数据集名字叫 `self.name` (后简称 `name`), 那么我们的 AutoBase 就会支持如下两种情形。

- 路径 `sdf/name` 是一个文件夹, 那么我们会认为该文件夹中有两个文件。
 - `train.txt/train.csv`: 训练数据集所对应的文件。
 - `test.txt/test.csv`: 交叉验证集所对应的文件。如果该文件不存在的话, 就认为用户不想使用交叉验证集, 而只想进行训练。
- 文件 `sdf/name.csv` 或 `sdf/name.txt` 存在, 那么我们会认为该文件就是完整的(即训练+交叉验证)数据集所对应的文件。

注意: 后缀究竟是 `txt` 还是 `csv`, 取决于 `file_type` 这个变量的具体取值。

在知道具体的功能之后, 相应的实现就比较自然了。

```

01 def get_data_from_file(self, file_type, test_rate, target=None):
02     # 如果文件后缀是 txt 的话, 就认为分隔符为空格, 且文件不含 header
03     if file_type == "txt":
04         sep, include_header = " ", False
05     # 如果文件后缀是 csv 的话, 就认为分隔符为逗号, 且文件含 header
06     elif file_type == "csv":
07         sep, include_header = ",", True

```

```

08     # 对于其他的文件后缀暂无实现，不过可以很方便地进行拓展
09     else:
10         raise NotImplementedError(
11             "File type '{}' not recognized".format(file_type))
12     # 如果没有提供目标文件格式数据的名字的话
13     if target is None:
14         # 就根据 self.data folder 和 self.name
15         # 来自动获取目标文件格式数据的名字
16         target = os.path.join(self.data_folder, self.name)
17     # 如果该名字没有对应一个文件夹的话
18     if not os.path.isdir(target):
19         # 就说明用户提供了完整的数据集，从而直接调用
20         # 第3章实现的 Toolbox.get_data 即可
21         with open(target + ".{}".format(file_type), "r") as file:
22             data = Toolbox.get_data(file, sep, include_header)
23     # 否则，就分别尝试读取训练集与交叉验证集
24     else:
25         # 利用 Toolbox.get_data 读入训练集
26         with open(os.path.join(target, "train.{}".format(file_type)), "r") as file:
27             train_data = Toolbox.get_data(file, sep, include_header)
28         # 检查交叉验证集是否存在
29         test_file = os.path.join(target, "test.{}".format(file_type))
30         # 如果不存在，则只使用训练集，且把 test_rate 设置为 0
31         if not os.path.isfile(test_file):
32             data, test_rate = train_data, 0
33         # 如果存在，则读入交叉验证集并把 data 设置为 tuple
34         else:
35             with open(test_file, "r") as file:
36                 test_data = Toolbox.get_data(file, sep, include_header)
37                 data = (train_data, test_data)
38     return data, test_rate

```

至此我们就完成了数据的读入部分，接下来要做的是从这些读进来的数据中自动提取出想要的信息。为方便大家理解，在此我们先把需要提取的信息做一个罗列。

- **self.numerical_idx**: 储存着各维特征的特征类型与标签类型的列表，在上一章有过相应的介绍。不过在本章中，其元素除了可能是 True 和 False 以外，还可能是 None。而当它的某个元素取 None 时，就意味着相应的特征是冗余特征。
- **self.feature_sets**: 记录各维特征的所有特征取值的列表。具体而言，假设第 i 维的特征有 a_1, a_2, \dots, a_{n_i} 这 n_i 个取值的话，那么 **self.feature_sets** 这个列表的第 i 个元素就是含 $a_1 \sim a_{n_i}$ 的一个 Python 集合 (set)。而如果第 i 维的特征是冗余特征或连续型特征的话，那么相应的第 i 个元素就是一个空集合。
- **self.n_features**: 记录各维特征的特征取值个数的列表，它能由 **self.feature_sets** 直接推得（比如在上面这个例子中，**self.n_features** 的第 i 个元素就是 n_i ）。
- **self.all_num_idx**: 记录各维特征是否是数值型特征的列表。如果某个维度的特征不是数值型特征的话，就需要先将其转换成数值特征，然后再做后续的处理。

注意：我们曾在 1.3.2 节中简要介绍过如何数值化数据，而在下一节中，我们则会介绍如何把（完整的）数据转换的过程自动化、可复用化。

不难看出，如果能获取以上四个信息的话，那么对于数据方方面面的特性，我们就都能有一个大致的把握了。同时由于提取信息的过程本身相对独立，所以我们应该把其核心过程抽象成一个单独的方法。由于该方法的实现需要用到许多 Python 的技巧，所以我们会把它的相关说明放在附录 F 中，感兴趣的读者可以先行阅读相应的部分，源代码则可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/NNUtil.py。

```

96         # 如果数据信息对应的缓存文件夹不存在，就建立一个缓存文件夹
97         if not os.path.isdir(data_info_folder):
98             os.makedirs(data_info_folder)
99         # 如果不存在保存数据信息的缓存文件或处于第 1 阶段的话
100        # 就需要从头开始提取数据信息
101        if not os.path.isfile(data_info_file) or stage == 1:
102            print("Generating data info")
103            # 如果提供了 numerical_idx
104            # 就把 self.numerical_idx 赋为相应的值
105            if numerical_idx is not None:
106                self.numerical_idx = numerical_idx
107            # 否则，如果 self.numerical_idx 不是 None 的话
108            # 就把 numerical_idx 赋为 self.numerical_idx
109            elif self.numerical_idx is not None:
110                numerical_idx = self.numerical_idx
111            # 如果数据的信息不齐全的话，就需要利用
112            # Toolbox.get_feature_info 这个方法来完成
113            # 提取数据信息的核心过程（其具体的实现说明会放在附录 F 中）
114            if not (self.feature_sets and self.n_features and self.all_num_idx):
115                # 从 self.data_info 中获取当前问题是否是回归问题的信息
116                is_regression = self.data_info.pop(
117                    "is regression",
118                    numerical_idx is not None and numerical_idx[-1]
119                )
120                info = Toolbox.get_feature_info(
121                    data, numerical_idx, is_regression)
122                self.feature_sets, self.n_features = info[:2]
123                self.all_num_idx, self.numerical_idx = info[2:]
124            # 如果 self.numerical_idx 的最后一位是 True
125            # 就意味着标签是连续的，即此时要做的是回归问题
126            # 所以就把 self.n_class 属性置为 1；否则，就把它设置为
127            # 标签的取值个数（即类别个数）
128            self.n_class = 1 if self.numerical_idx[-1] else self.n_features[-1]
129            # 调用 self._get_transform_dicts 方法来获取
130            # 进行数据转换时需要用到的字典
131            self.get_transform_dicts()
132            # 将提取出来的数据信息写入缓存文件
133            with open(data_info_file, "wb") as file:
134                pickle.dump([

```

```

135         self.n_features, self.numerical_idx, self.transform_dicts
136     ], file)
137     # 如果存在数据信息相应的缓存文件且处于第3阶段的话
138     # 就从相应的缓存文件中读出数据信息
139     elif stage == 3:
140         print("Restoring data info")
141         with open(data_info_file, "rb") as file:
142             info = pickle.load(file)
143             self.n_features, self.numerical_idx, self.transform_dicts = info
144             # 用相同的逻辑给 self.n_class 赋值
145             self.n_class = 1 if self.numerical_idx[-1] else self.n_features[-1]

```

至此我们就完成了数据信息提取的部分，而接下来要做的，就是根据这些数据信息来推算出其余的数据特征，然后进行相应的数据转换。

```

146     # 如果数据不是从缓存文件读取而来的话，就根据
147     # n_train（训练集样本数）的取值来划分训练集与交叉验证集
148     if not use_cached_data:
149         if n_train != -1:
150             train_data, test_data = data[:n_train], data[n_train:]
151             # 如果 n_train 为-1的话，由前文可知此时将不使用交叉验证集
152             # 所以就把 test_data 设置为 None
153         else:
154             train_data, test_data = data, None
155             # 获取训练集、交叉验证集的名字
156             train_name, test_name = names
157             # 调用 self._transform_data 方法，完成数据的转换
158             # 该方法会在下一节中进行介绍，这里暂时按下不表
159             train_data = self._transform_data(
160                 train_data, train_name, train_name, True, True, stage)
161             if test_data is not None:
162                 test_data = self._transform_data(
163                     test_data, test_name, train_name, True, stage=stage)
164             # 调用 self._gen_categorical_columns 方法
165             # 自动生成 self.categorical_columns 这个属性
166             self._gen_categorical_columns()
167             # 如果处于第3阶段，而且数据不是从缓存文件中读取的话
168             # 就把处理好的数据写入缓存文件，以方便下次运行程序时复用
169             if not use_cached_data and stage == 3:
170                 print("Caching data...")
171                 if not os.path.exists(data_cache_folder):
172                     os.makedirs(data_cache_folder)
173                 np.save(train_data_file, train_data)
174                 if test_data is not None:
175                     np.save(test_data_file, test_data)
176
177             # 获取训练、交叉验证所用的特征向量与标签
178             x, y = train_data[..., :-1], train_data[..., -1]
179             if test_data is not None:
180                 x_test, y_test = test_data[..., :-1], test_data[..., -1]

```

```

181         else:
182             x_test = y_test = None
183             # 计算特征向量的稀疏性
184             # 在此我们认为, 0 与 nan 都是会贡献稀疏度的元素
185             self.sparsity = ((x == 0).sum() + np.isnan(x).sum()) / np.prod(x.shape)
186             # 计算标签中各个类别出现的次数
187             , class_counts = np.unique(y, return counts=True)
188             # 利用第2章介绍朴素贝叶斯时的算法
189             # 计算出各个类别的先验概率的极大似然估计
190             self.class_prior = class_counts / class_counts.sum()
191
192             # 把自动生成的 numerical_idx 与 categorical_columns 的信息
193             # 赋给 self.data_info 这个管理数据超参数的字典
194             self.data_info["numerical_idx"] = self.numerical_idx
195             self.data_info["categorical_columns"] = self.categorical_columns
196
197             return x, y, x_test, y_test

```

其中在第166行用到的 `self.gen_categorical_columns` 方法的实现, 需要用到5.2.3节中初始化 `AdvancedNN` 时定义过但却一直没有用到过的“真正有效的 `numerical_idx`”——`self.valid_numerical_idx` 这个属性了。由当时的定义可知, 该属性会返回 `numerical_idx` 中的非 `None` 元素。而由上文的讨论可知, `numerical_idx` 中如果出现了 `None` 的话, 就意味着对应的特征是冗余特征。所以 `valid_numerical_idx` 的意义就很明确了: 它是记录非冗余特征的特征类型的列表。

类似的, 我们还需要定义出记录非冗余特征的特征取值个数的列表——`valid_n_features`:

```

01     @property
02     def valid_n_features(self):
03         return np.array([
04             n_feature for i, n_feature in enumerate(self.n_features)
05             if self.numerical_idx[i] is not None
06         ])

```

而在有了 `valid_numerical_idx` 和 `valid_n_features` 之后, 根据 `categorical_columns` 的定义来进行相应的推算就比较容易了:

```

07     def gen_categorical_columns(self):
08         # 利用真正有效的 numerical_idx 和 n_features 属性
09         # 来直接推出 categorical_columns 这个属性的取值
10         self.categorical_columns = [
11             (i, value) for i, value in enumerate(self.valid_n_features)
12             if not self.valid_numerical_idx[i] and
13             self.valid_numerical_idx[i] is not None
14         ]
15         # 如果标签不是连续型标签的话
16         # 那么上述过程会把标签的相关信息也记录进 categorical_columns 中
17         # 所以就需要把 categorical_columns 的最后一个元素 pop 出来
18         if not self.valid_numerical_idx[-1]:
19             self.categorical_columns.pop()

```

至此，我们就完成了所有的数据准备工作。在下一节中，我们将会详细介绍上述代码中没有展开说明的和数据转换有关的部分。

6.2 数据的转换

在 6.1 节中，我们把数据准备的整个流程都过了一遍，不过其中许多关键的和数据转换相关的步骤都还没有展开叙述。在这一节中，我们则会一一对它们进行补充说明。

由 6.1.2 节可知，数据的转换主要包含如下两步：

- 非数值型离散型特征的数值化与冗余特征的去除（第 1 阶段、第 3 阶段）。
- 缺失值处理与连续型特征的预处理（第 2 阶段、第 3 阶段）。

此外在做完数据转换后，我们可能还需要对一些特殊的情况做进一步的处理。所以在 6.2.1 和 6.2.2 节中，我们会对第 1 步的具体做法进行说明；而在 6.2.3 和 6.2.4 节中，我们则会对第 2 步的具体做法进行说明。在 6.2.5 节中，我们则会说明如何处理两种特殊类型的数据。

6.2.1 数据的数值化

首先是 AutoBase 中的 `self._get_transform_dicts` 方法，它能生成一个帮助我们将非数值型特征数值化的列表——`self.transform_dicts`。如果对 1.3.2 节尚有印象的话，那么就不难知道 `self.transform_dicts` 应该是一个储存了很多“转换字典”的列表，其中第 i 个转换字典是能将第 i 维特征数值化的转换字典。不过，由于原始的数据可能已经是数值型特征，而且还可能会有连续型特征、冗余特征与特征中的缺失值等，所以我们需要对这些情况进行特殊的处理。具体而言：

- 如果第 i 个特征是连续型特征，那么 `self.transform_dicts` 的第 i 个元素就是空字典。
- 如果第 i 个特征是离散型的数值型特征的话，要分以下两种情况讨论。
 - 如果该特征的特征取值是连续的 n 个非负整数，且取值从 0 开始（即其特征取值为 $0, 1, \dots, n-1$ ）的话，那么就不用进行额外的数值化，所以 `self.transform_dicts` 的第 i 个元素也是空字典。
 - 否则，由于做 Embedding 时要求输入的离散型特征的取值需要是从 0 开始的 n 个非负整数，所以还是需要进行（配合 Embedding 的）数值化。
- 如果第 i 个特征是冗余特征的话，就把 `self.transform_dicts` 的第 i 个元素设为 `None`。
- 如果第 i 个特征含有缺失值（nan），但是却需要进行转换的话，就把字符串“nan”作为 `self.transform_dicts` 的第 i 个元素对应的字典的键值（key）。

相应的代码实现如下所示。

```

01     def get_transform_dicts(self):
02         self.transform_dicts = [
03             # 如果 numerical_idx 为 None, 说明相应特征是冗余特征
04             # 所以把 self.transform_dicts 中的相应元素设为 None
05             None if is_numerical is None else
06             # 如果原始特征的特征取值是离散型特征, 而且其特征取值不是
07             # 从 0 开始的连续 n 个非负整数的话
08             # 就需要把相应的转换字典算出来
09             {key: i for i, key in enumerate(sorted(feature_set))}
10             if not is_numerical and (not all_num or not np.allclose(
11                 np.sort(np.array(list(feature_set), np.float32).astype(np.int32)),
12                 np.arange(0, len(feature_set)))
13             # 否则, 就把相应元素设置为空字典即可
14             ) else {} for is_numerical, feature_set, all_num in zip(
15                 self.numerical_idx[:-1], self.feature_sets[:-1],
16                 self.all_num_idx[:-1]
17             )
18         ]
19         # 如果是回归问题, 说明标签肯定是连续型标签
20         # 所以在 self.transform_dicts 中加入一个空字典即可
21         if self.n_class == 1:
22             self.transform_dicts.append({})
23         # 否则, 就调用 self._get_label_dict 方法来获取
24         # 用于转换标签的转换字典
25         else:
26             self.transform_dicts.append(self._get_label_dict())

```

其中, 第 26 行调用的 `self._get_label_dict` 方法是比较直观的:

```

01     def get_label_dict(self):
02         # 利用 self.feature_sets 获取标签的所有取值
03         labels = self.feature_sets[-1]
04         # 对这些标签排序, 存储在一个列表中
05         sorted_labels = sorted(labels)
06         # 如果标签并非全是数值型标签的话
07         if not all(Toolbox.is_number(str(label)) for label in labels):
08             # 就直接返回相应的转换字典
09             return {key: i for i, key in enumerate(sorted_labels)}
10         # 否则, 就要判断是否需要转换数值型的标签
11         numerical_labels = np.array(sorted_labels, np.float32)
12         if numerical_labels.max() - numerical_labels.min() != self.n_class - 1:
13             return {key: i for i, key in enumerate(sorted_labels)}
14         return {}

```

在上述代码的第 7 行里, 我们调用了 `Toolbox.is_number` 方法, 该方法能判断一个字符串是否是数值型的字符串, 它的实现需要用到 Python 的一些小技巧, 感兴趣的读者可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/NNUtil.py。

6.2.2 冗余特征的去除

在完成了数值化数据的工作后，我们就能着手去除冗余的特征了。虽说判断一个特征是否是冗余特征是非常困难的事情，但至少在如下两种情况下我们是能比较确信相应的特征是冗余特征的：

- 该特征全是整数取值而且各不相同，此时该特征一般是各个样本的 id，它没有实际的意义，所以可以把它视为冗余特征。
- 该特征的特征取值个数只有 1 个，此时该特征自然是冗余特征。

所以在实现 `Toolbox.get_feature_info` 这个提取数据信息的方法时，我们会充分考虑这两种情况，并把符合这两种情况的特征标记为冗余特征。而如果假设已经标记好了冗余特征的话，把它们去除这一步是比较简单的。下面我们就结合上一节叙述的数值化数据的步骤，来看看去除冗余特征的具体代码实现。

```

01     """
02     数据转换的核心方法
03     data: 数据集
04     name: 数据集 data 的名字
05     train_name: 训练集的名字
06     include_label: 代指数据集 data 是否含有标签
07     refresh_redundant_info: 代指是否刷新冗余特征的信息
08     我们会用 self.whether_redundant 属性来储存冗余特征的信息
09     stage: 数据预处理所处的阶段
10     """
11     def transform_data(self, data, name, train_name="train",
12                        include_label=False, refresh_redundant_info=False, stage=3):
13         # 判断 data 是否是 numpy 数组，并以此调整数据转换的行为
14         is_ndarray = isinstance(data, np.ndarray)
15         # 如果需要刷新冗余特征的信息或者还没有冗余特征的信息的话
16         if refresh_redundant_info or self.whether_redundant is None:
17             # 根据转换字典列表来判断各特征是否是冗余特征
18             # 由前文的讨论可知，如果某个转换字典是 None
19             # 那么相应的特征就是冗余特征
20             self.whether_redundant = np.array([
21                 True if local_dict is None else False
22                 for local_dict in self.transform_dicts
23             ])
24             # 根据冗余特征的信息和其他数据信息来判断
25             # 哪些特征是需要进行数值化的特征，
26             # 并把可能用到的信息记录下来
27             targets = [
28                 (i, local_dict) for i, (idx, local_dict) in enumerate(
29                     zip(self.numerical_idx, self.transform_dicts)
30                 ) if not idx and local_dict and not self.whether_redundant[i]
31             ]
32             # 如果需要进行数值化
33             # 而且 targets 中含有标签相关的信息

```



```

34     # 但是数据本身不含有标签的话
35     # 就要把标签的信息（最后一个元素）去除
36     if targets and targets[-1][0] == len(self.numerical_idx) - 1
37         if not include_label:
38             targets = targets[:-1]
39     # 如果处于第1阶段或第3阶段的话
40     # 就进行数据的数值化工作与冗余特征的去除
41     if stage == 1 or stage == 3:
42         # 看一下是否有冗余特征
43         n_redundant = np.sum(self.whether_redundant)
44         if n_redundant == 0:
45             whether_redundant = None
46         else:
47             whether_redundant = self.whether_redundant
48             # 因为标签不能是冗余特征，所以如果数据集含有标签的话
49             # 就不取冗余特征信息的最后一个元素
50             if not include_label:
51                 whether_redundant = whether_redundant[:-1]
52     # 生成不冗余的特征的下标
53     valid_indices = [
54         i for i, redundant in enumerate(self.whether_redundant)
55         if not redundant
56     ]
57     # 因为标签肯定是不冗余的，所以如果数据不含标签的话
58     # 就要去除掉最后一个元素
59     if not include_label:
60         valid_indices = valid_indices[:-1]
61     # 对数据集 data 中的每一行都进行相应的数值化
62     for i, line in enumerate(data):
63         for j, local_dict in targets:
64             elem = line[j]
65             # 如果相应的数据是字符串，就用转换字典进行转换
66             # 而如果该数据不在转换字典中，则尝试用 nan（缺失值）
67             # 来代指它；而如果连 nan 都不在转换字典中
68             # 则统一把该数据转换为“转换字典的长度”这个数值
69             if isinstance(elem, str):
70                 line[j] = local_dict.get(
71                     elem, local_dict.get("nan", len(local_dict)))
72             # 如果相应的数据是 nan（缺失值）
73             # 就读取转换字典中"nan"对应的 value 来转换
74             elif math.isnan(elem):
75                 line[j] = local_dict["nan"]
76             # 对于其余的情况，也用转换字典直接进行类似的转换
77             else:
78                 line[j] = local_dict.get(
79                     elem, local_dict.get("nan", len(local_dict)))
80     # 如果 data 不是 numpy 数组，而且有冗余特征的话
81     # 就在 for 循环内部去除冗余的特征
82     if not isinstance(data, np.ndarray) and whether_redundant is not None:

```

```

83         data[i] = [line[j] for j in valid indices]
84     # 如果 data 是 numpy 数组, 而且有冗余特征的话
85     # 就利用 numpy 的索引功能, 直接去除冗余的特征
86     if is ndarray and whether redundant is not None:
87         data = data[..., valid indices].astype(np.float32)
88     # 否则, 就把 data 转换成 numpy 数组以方便后续的操作
89     else:
90         data = np.array(data, dtype=np.float32))

```

以上就是数据转换第 1 步的相关实现。在运行完上述代码后, 得到的 `data` 就已经是数值化的、没有冗余特征的原始数据了。而对于第 2 步中的缺失值处理与连续型特征的数据预处理而言, 由于它们都是相对独立的过程, 所以为了让代码结构更加合理、优雅, 把它们都抽象成单独的类是有必要的。我们会在 6.2.3 节和 6.2.4 节中分别介绍它们的具体实现, 这里就先看一下如何将它们整合进 `self._transform_data` 这个方法中。

```

91     # 如果处于第 2 阶段或第 3 阶段的话
92     # 就进行数据的进一步处理
93     if stage == 2 or stage == 3:
94         data = np.asarray(data, dtype=np.float32)
95         # 实例化一个“缺失值处理器”(NanHandler), 其中:
96         # 1) self.nan_handler_method 代指缺失值的处理方法
97         # 2) self.reuse_nan_handler_values 代指测试时的处理手段
98         # 这两个属性的具体含义会在 6.2.3 节进行相应的说明
99         if self._nan_handler is None:
100             self._nan_handler = NanHandler(
101                 method=self.nan_handler_method,
102                 reuse_values=self.reuse_nan_handler_values
103             )
104         # 利用该实例的 transform 方法, 完成缺失值的处理
105         data = self._nan_handler.transform(
106             data, self.valid_numerical_idx[:-1])
107         # 这里的 self._pre_processors 是 None 或字典
108         # 如果它不是 None, 就意味着需要进行连续型特征的数据预处理
109         if self._pre_processors is not None:
110             # 根据 self.reuse_mean_and_std 属性的取值
111             # 决定将要使用的“数据预处理器”(PreProcessor) 的名字
112             # 关于 self.reuse_mean_and_std 属性的具体含义
113             # 我们会在 6.2.4 节进行相应的说明
114             if self.reuse_mean_and_std:
115                 pre_processor_name = train_name
116             else:
117                 pre_processor_name = name
118             # 然后实例化一个“数据预处理器”(PreProcessor)
119             # 并以名字为键值(key)、实例为值(value)
120             # 更新 self._pre_processors 这个字典以方便后续的复用
121             pre_processor = self._pre_processors.setdefault(
122                 pre_processor_name, PreProcessor(
123                     self.pre_process_method, self.scale_method
124                 )

```

```

125         )
126         # 如果 data 不含标签, 就利用该实例的 transform 方法和
127         # 有效的 numerical idx 来对整个 data 进行数据预处理
128         if not include label:
129             data = pre_processor.transform(
130                 data, self.valid_numerical_idx[:-1])
131         # 否则, 就排除 data 的最后一列
132         # 因为我们一般不会对标签进行预处理
133         else:
134             data[..., :-1] = pre_processor.transform(
135                 data[..., :-1], self.valid_numerical_idx[:-1])
136         return data

```

可以看到, 相应的实现还是比较直观的。其中, 数据预处理能够得以复用的核心就在于我们会维护 `self_pre_processors` 这个字典, 它储存着各式各样的“数据预处理器”以供我们在新的数据集上进行与训练集相同的数据预处理过程。

6.2.3 缺失值处理

本节我们会介绍如何处理数据中的缺失值以及缺失值处理器——`NanHandler` 的具体实现。虽然在此之前我们用的数据集都非常“漂亮”——它们每个样本中的特征向量都是完整的、没有缺失值的, 但是在现实任务中, 常常会由于统计失误、源数据丢失等问题, 导致数据集中的某些特征向量可能并不完整。此时, 这些特征向量中某些维度的特征在文件中可能就会以“nan” (not a number 的缩写) 的形式出现。如果直接把 nan 喂给机器学习模型的话, 除非该模型天生适应于特征向量中含有 nan 的情形 (比如 `xgboost`), 否则可能会直接导致模型的训练崩溃。以神经网络为例, 如果把 nan 输入模型的话, 那么反向传播时的梯度就都会是 nan, 从而导致所有的参数都会变成 nan。

注意: 在后续的讨论中, 我们会默认缺失值都以 nan 的形式出现。

因此对缺失值 nan 进行特殊的处理是很有必要的。在 6.2.1 节中, 我们其实已经介绍了如何处理离散型特征中的 nan——只需把它看成另一种特征取值即可。具体而言, 如果第 i 维的特征本身有 n_i 个取值 ($a_1 \sim a_{n_i}$), 而且又有缺失值 nan 的话, 那么我们就可以把所有的 nan 都视为第 i 维特征的第 $n_i + 1$ 个取值。所以对于 `NanHandler` 这个类而言, 它只需要关心如何处理连续型特征中的 nan 即可。

不过, 由于缺失值处理本身是很难的问题, 相应的解决方案也五花八门, 所以我们无法在一节中把所有的处理手段都说清楚。为此, 我们接下来只会介绍其中比较常用的两种, 具体如下所述。

- 直接把所有特征向量中含 nan 的样本去除 (delete)。当样本数量众多, 而 nan 的占比相对较少时, 这种方法是简单而有效的。
- 用第 i 维特征的某个统计量来替换第 i 维特征中的 nan, 比如用均值或中位数。
 - 均值 (mean): 当第 i 维特征的分布比较像正态分布时, 使用 mean 是比较合理的。

- 中位数 (median)：当第 i 维特征存在一些异常值时，使用 median 会比使用 mean 要更合理，因为 median 不受个别的异常值影响，而 mean 可能会受比较大的影响。

其中，第一种处理方法比较好理解，实现起来也没有太过特殊的地方，但是第二种处理方法则还需要额外考虑一个问题：模型在进行测试时，第 i 维特征中的 nan 是直接测试集的统计量还是复用训练集的统计量？一般来说，如果测试集样本量充足的话，直接用测试集的统计量是一个不错的选择；然而如果测试集样本很少，甚至说只有单一样本时，使用测试集的统计量显然就不尽合理。为此，如果采用第二种处理方法的话，实现时就要允许用户进行相应的选择。

注意：像这种测试集需要额外进行考虑的情形在 6.2.4 节讨论连续型特征数据预处理时也会遇到，而这也正是为何我们需要定义 self.reuse_mean_and_std 属性的原因。

综上所述，在 6.2.2 节的代码中实例化 NanHandler 时，曾经使用到的两个属性的具体含义就比较清晰了。

- self.nan_handler_method：缺失值处理的方法，即 delete、mean 和 median。
- self.reuse_nan_handler_values：模型在测试时是否复用训练集的统计量。

同时，NanHandler 本身的代码实现也就比较直观了，如代码 6.1 所示。

代码 6.1 缺失值处理器的实现：NNUtil.py

```
01 class NanHandler:
02     """
03     初始化结构
04     method: 缺失值处理的方法
05     reuse values: 测试时是否复用训练集的统计量
06     """
07     def __init__(self, method, reuse_values=True):
08         # 使用 self.values 属性来记录训练集的各个统计量
09         self.values = None
10         self.method = method
11         self.reuse_values = reuse_values
12
13     # 缺失值处理的具体方法
14     # 参数 refresh values 代指“是否刷新统计量”
15     def transform(self, x, numerical_idx, refresh_values=False):
16         # 如果缺失值处理方法为 None 的话，就意味着什么也不做
17         if self.method is None:
18             Pass
19         # 如果是 delete 的话，就把存在 nan 的样本都删除
20         # 这里利用了“~”以及 np.any，我们会在后文进行相应的说明
21         elif self.method == "delete":
22             x = x[~np.any(np.isnan(x[... , numerical_idx]), axis=1)]
23         # 否则，就利用统计量来替换 nan
24         else:
25             # 如果 self._values 是 None，就进行初始化
26             if self._values is None:
```

```

27         self.values = [None] * len(numerical_idx)
28     # 进入缺失值处理的主循环
29     for i, numerical in enumerate(numerical_idx):
30         v = self.values[i]
31         # 注意我们只关心连续型特征的缺失值处理
32         # 所以如果当前特征不是连续型的，就直接跳过
33         if not numerical:
34             continue
35         # 提取出第 i 维特征
36         feat = x[:, i]
37         # 利用 np.isnan 获取第 i 维特征中 nan 的位置
38         mask = np.isnan(feat)
39         # 如果“没有任意一个 nan”的话
40         # 就说明第 i 维特征不用进行缺失值处理
41         if not np.any(mask):
42             continue
43         # 如果（测试时）已有相应的（训练集的）统计量
44         # 而且属性设置为复用（训练集的）统计量
45         # 而且参数设置为不刷新统计量的话
46         if v is not None and self.reuse_values and not refresh_values:
47             # 就把相应的统计量设置为“目标”
48             new_value = v
49         else:
50             # 否则，利用 getattr 和 self.method 来计算相应的统计量
51             # 并在把该统计量记录到 self.values 后
52             # 把该统计量设置为“目标”
53             new_value = getattr(np, self.method)(feat[~mask])
54             if self.reuse_values and (v is None or refresh_values):
55                 self.values[i] = new_value
56             # 用“目标”替换掉第 i 维特征中 nan 的位置
57             feat[mask] = new_value
58     return x

```

其中，在上述代码的第 22 行、第 41 行处我们用了 `np.any`，这个方法的作用和它的名字完全一致：只要某个数组 `arr` 中含有一个 `True`，那么 `np.any(arr)` 就会返回 `True`；只有 `arr` 中一个 `True` 都没有时，`np.any(arr)` 才会返回 `False`。注意到第 22 行调用 `np.any` 时还加了一个 `axis = 1` 的参数，这和 `np.mean`、`np.median` 之类的方法中的 `axis` 参数是一致的。具体而言，假设 `arr` 是一个 $N \times n$ 的二维数组，那么 `np.any(arr, axis=1)` 就会返回一个含 N 个元素的数组 `ans`，且：

- 当 `arr` 的第 i 行有一个 `True` 时，`ans` 的第 i 个元素就是 `True`。
- 当 `arr` 的第 i 行全是 `False` 时，`ans` 的第 i 个元素才是 `False`。

所以第 22 行中的

```
np.any(np.isnan(x[:, numerical_idx]), axis=1)
```

其实就是把数据集 `x` 的连续型特征的部分 (`x[:, numerical_idx]`) 中含 `nan` 的行的位置给识别出来。如果 `x` 的第 i 行中有 `nan` 的话，那么这行代码返回的数组的第 i 个元素就会是 `True`。

而上述代码的第 22 行、第 53 行处我们用了“~”这个符号。该符号的意义是“逻辑非”，实际的作用是把某个逻辑数组（即元素全是 True 或 False 的数组）中的 True 变成 False、False 变成 True。因此如果 x 的第 i 行一个 nan 都没有的话，那么第 22 行中

```
~np.any(np.isnan(x[..., numerical_idx]), axis=1)
```

返回的数组的第 i 个元素就是 True，从而

```
x = x[~np.any(np.isnan(x[..., numerical_idx]), axis=1)]
```

返回的就是特征向量里一个 nan 都没有的样本的数据集，也就是使用 delete 方法所想要拿到的最终数据集。同理在第 53 行处，假设我们使用了 mean 作为缺失值处理的方法，那么

```
getattr(np, self.method)
```

就会返回 np.mean（计算均值的方法），从而

```
getattr(np, self.method)(feat[~mask])
```

返回的就是没有 nan 的 feat（第 i 维特征）的均值（因为 mask 记录着 nan 的位置）。

6.2.4 连续型特征的数据预处理

本节我们会介绍如何对数据中的连续型特征进行预处理（后简称“连续型预处理”），以及数据预处理器——PreProcessor 的基本思想。与缺失值处理类似，连续型预处理的解决方案也非常多，具体哪个更好也是需要结合具体问题具体分析，所以我们在一节范围内就只能抓住其中最常用的来说明。具体而言，我们所将采用的预处理的核心正是之前反复应用到的预处理方法：标准化（Normalization）。不过由于需要考虑到可复用性、可拓展性以及一些特殊的情况，所以在实现的层面上会有许多需要调整的地方。

首先是可复用性，这一点在 6.2.2 节中实现 self_transform_data 时就已经有过相应的考虑。具体而言，我们会维护一个叫 self_pre_processors 的字典，它存储着各式各样的数据预处理器，从而如果想复用训练时做的数据预处理过程的话，我们就只需把训练时用的数据预处理器存进这个字典，然后对这个字典进行保存与复用即可。

然后是可拓展性，为此我们需要梳理一下对连续型特征进行数据预处理的一般步骤：

- 发现特征中的异常值并对它们进行处理。
- 计算处理完异常值的特征的某些统计量。
- 利用这些统计量进行具体的预处理过程。

比如说对于标准化而言，由于其公式为

$$\phi_0(\mathbf{x}) = \frac{\mathbf{x} - \text{mean}}{\text{std}}$$

所以不难看出，如果特征 \mathbf{x} 中大部分数都比较小，而有某些数特别大的话，就会导致其标准差（std）非常大，从而使得预处理过后的 $\phi_0(\mathbf{x})$ 的变化非常小。比如，假设

$$\mathbf{x} = (x_1, x_2, \dots, x_{100}, x_{101})^T$$

且

$$\sum_{i=1}^{100} x_i = 1, \quad x_{101} = 100$$

以及

$$x_i \in [0, 0.1], \quad \forall i$$

的话，那么就有

$$\text{mean} = \frac{1}{101} \sum_{i=1}^{101} x_i = 1$$

以及

$$\begin{aligned} \text{std} &= \sqrt{\frac{1}{100} \sum_{i=1}^{101} (x_i - \text{mean})^2} = \sqrt{\frac{1}{100} \sum_{i=1}^{101} (x_i - 1)^2} \\ &\approx \sqrt{\frac{1}{100} (100 \times 0.95^2 + 99^2)} \approx 9.95 \end{aligned}$$

可以看到，标准差 std 和 $x_1 \sim x_{100}$ 之间差了将近两个量级（即 100 倍左右），这意味着标准化后的 $\phi_0(x_1) \sim \phi_0(x_{100})$ 彼此之间就会非常接近。所以对于标准化这个数据预处理方法而言，它在上述数据预处理一般步骤下的具体表现如下所述。

- 对特征中绝对值非常大的数进行一定的处理，如下所述。
 - 截断法（truncate）：设置一个上限 ϵ ，把绝对值超过该上限的取值设置为该上限：

$$f(\mathbf{x}) = \text{sign}(\mathbf{x}) \cdot \min(|\mathbf{x}|, \epsilon), \quad \epsilon > 0$$

当异常值较少时，这种做法是比较合理的。

- 成倍缩小法（divide）：将该特征的所有取值都同时除以某个数 ϵ ：

$$f(\mathbf{x}) = \text{sign}(\mathbf{x}) \cdot \frac{|\mathbf{x}|}{\epsilon}, \quad \epsilon > 0$$

当该特征整体取值都偏大时，这种做法是比较合理的。

- 取对数法（log）：对该特征的所有取值都取一个对数：

$$f(\mathbf{x}) = \text{sign}(\mathbf{x}) \cdot \log|\mathbf{x}|$$

当该特征本身具有对数意义时，这种做法是比较合理的。

- 计算处理完异常值的特征的均值（mean）与标准差（std）。
- 进行特征的标准化：

$$\phi_0(\mathbf{x}) = \frac{\mathbf{x} - \text{mean}}{\text{std}}$$

其中，虽然标准化本身的实现不难（可以参见 1.3.2 节），但是处理异常值时却需要许多

numpy 的技巧来保证运行速度。由于相应的代码实现比较冗长，所以大家只需熟悉上述这些步骤即可，感兴趣的读者也可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/NNUtil.py 中最后的 PreProcessor 部分。

6.2.5 特殊类型数据的处理

在之前的 4 节中，我们介绍了许多数据转换的方法，其中也曾提到需要对某些特征的特殊取值进行特别的处理（比如标准化中绝对值很大的特征取值）。在这一节中，我们会主要针对两种特殊类型的数据进行讨论：不平衡数据与稀疏数据。

首先来看看不平衡数据 (Imbalanced Data)。所谓的不平衡数据，一般是指在二分类问题中，类别为 0 的样本要比类别为 1 的样本多很多。对于多分类问题而言，我们可以类似地进行定义：如果出现次数最多的类别样本数比出现次数最少的类别样本数多很多，那么就称该数据为不平衡数据。

为什么需要处理不平衡数据呢？因为我们在不平衡数据上进行训练时，出现次数少的类别通常会被“忽视”。以二分类为例，假设现在有 999 个类别为 0 的样本和 1 个类别为 1 的样本，那么模型在训练时看到的样本就几乎全是类别为 0 的样本，从而它的输出也就会几乎全是 0。然而我们知道，通常在这种情况下，类别为 1 的样本才是最为关键的样本。比如我们想用机器学习模型去各个地方找金矿，可能实际情况下 1000 个地方才能有 1 个，但这一个却是最为关键的。如果不对不平衡数据做处理的话，那么模型就永远都输出 0（即“此处没有金矿”），从而我们就永远都找不到金矿了。

那么究竟应该如何处理不平衡数据呢？事实上相应的处理方式有很多，它们大致可以分为如下两种。

- 在数据层面上做处理，这种方式一般表现为各种采样的手段，包括但不限于：
 - Over Sampling（可翻译成“过采样”），它会在模型训练时增大选取少数派类别对应的样本的概率。
 - Under Sampling（可翻译为“欠采样”），它会在模型训练时减少选取多数派类别对应的样本的概率。
 - SMOTE (Synthetic Minority Oversampling Technique)，可参见 <https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume16/chawla02a-html/chawla2002.html>。
 - ADASYN (Adaptive Synthetic Sampling Approach)，可参见 <http://sci2s.ugr.es/keel/pdf/algorithm/congreso/2008-He-ieee.pdf>。
 - Informed Under-Sampling，可参见 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.252.8103&rep=rep1&type=pdf>。
- 在算法层面上做处理，比如即将介绍的调整各个样本的权重这个方法。

虽说这些手段有很多，但由于我们的目的是完成半自动化的机器学习框架 (AutoBase)，所以暂时只需实现一个效果不错的方法即可。具体而言，我们会通过调整各个样本的样本权重来

完成不均衡数据的处理。注意到之前定义代价函数时，其公式为

$$C(D) = \frac{1}{N} \sum_{i=1}^N L(G(\mathbf{x}_i), y_i)$$

其中， G 是我们的模型， (\mathbf{x}_i, y_i) 是数据集 D 中的样本。而之前没有指出的是，这个公式事实上认为所有样本都是“同样重要的”。换句话说，我们认为这 N 个样本的样本权重都是 $1/N$ 。对于更一般的情形而言，假设第 i 个样本的样本权重是 w_i ，那么上面代价函数的公式就应该改写为：

$$C(D) = \sum_{i=1}^N w_i \cdot L(G(\mathbf{x}_i), y_i)$$

那么在对某个参数 θ 求梯度时，就有

$$\frac{\partial C}{\partial \theta} = \sum_{i=1}^N w_i \cdot \frac{\partial}{\partial \theta} L(G(\mathbf{x}_i), y_i)$$

换句话说，第 i 个样本带来的梯度的权重，或说“重要性”，也会是 w_i 。所以如果我们让不均衡数据中出现次数少的类别样本的权重增大的话，在利用梯度下降法进行训练时，它的梯度的重要性也就会增大。换句话说，此时我们就更看重出现次数少的类别，从而缓解不均衡数据可能带来的问题。

注意：不难看出，从理论上来说，这种调整样本权重的方法，是和数据层面上的过采样与欠采样方法等价的。

不过，即使是调整样本权重这单一的方法，其实也有很多不同的实现方式。囿于篇幅，我们在此就只介绍其中一种简明有效的实现：利用各个类别先验概率的极大似然估计来调整样本权重。具体而言，假设一共有 K 个类别 $c_1 \sim c_K$ ，它们的先验概率估计为 $p_1 \sim p_K$ ，再假设第 i 个样本属于第 k_i 类，即

$$y_i = c_{k_i}, \quad i = 1, 2, \dots, N$$

那么就令

$$w_i = \frac{1}{p_{k_i} \cdot \sum_{j=1}^K \frac{1}{p_j}}, \quad i = 1, 2, \dots, N$$

此时不难看出

$$w_1 \cdot p_{k_1} = w_2 \cdot p_{k_2} = \dots = w_N \cdot p_{k_N}$$

以及

$$\sum_{i=1}^N w_i = 1$$

即此时的 $w_1 \sim w_N$ 确实构成了样本权重，且各个样本的样本权重和它们所属类别的先验概率

成反比关系。

由于我们在第 3 章实现 TensorFlow 模型基本框架时已经完成了样本权重的相应实现并留下了一个待定的属性（`self._sample_weights`），所以这里只需算出相应的样本权重并给该属性赋值即可：

```
01 def handle_unbalance(self, y):
02     # 如果是回归问题，那就不存在上述的不均衡问题
03     # 所以就不进行任何操作
04     if self.n_class == 1:
05         return
06     # 算出各个类别先验概率（self.class_prior）中
07     # 最多的和最少的之间的比值
08     class_ratio = self.class_prior.min() / self.class_prior.max()
09     # 如果最多的和最少的之间相差 10 倍以上
10     # 而且用户没有事先给定样本权重的话
11     if class_ratio < 0.1 and self._sample_weights is None:
12         # 就按照上述公式计算出各个样本的样本权重
13         self._sample_weights = np.ones(
14             len(y)) / self.class_prior[y.astype(np.int)]
15         self._sample_weights /= self._sample_weights.sum()
16         self._sample_weights *= len(y)
```

注意：虽然回归问题不存在“某些类别对应的样本有很多，而另一些类别的样本很少”这样的不均衡问题，但它却存在着“某些样本的标签绝对值特别大，而另一些样本的标签绝对值较小”这样的不均衡问题。由于回归问题并非是神经网络擅长的领域，所以我们就不在此展开叙述这种不均衡回归问题的解决方案了，

然后来看稀疏数据的问题。所谓的稀疏数据，一般代指的是两种情形下的数据：

- 数据的特征本身含有非常多的 0。
- 数据含有非常多的缺失值。

而事实上在 6.1.2 节中计算数据稀疏性时，我们就把数据中取值为 0 的元素和取值为 nan 的元素的比例作为数据的稀疏度。

稀疏数据的处理方法也有很多，比如 PCA (Principal Components Analysis)、EFA (Exploratory Factor Analysis)、CFA (Confirmatory Factor Analysis) 等（可参见 <https://stats.stackexchange.com/questions/91301/confirmatory-factor-analysis-for-data-reduction-prior-to-regression/96160#96160> 与 <https://stats.stackexchange.com/questions/126882/how-to-build-a-predictive-model-with-a-billion-of-sparse-features>）。毫不夸张地说，稀疏数据，尤其是大规模稀疏数据的学习是一个非常大的领域，在这里不能进行面面俱到的介绍，为此我们只针对神经网络这个模型来对稀疏数据做一个直观的说明。

在第 5 章介绍 Dropout 时曾经说过，Dropout 会在每次训练中随机丢弃一些神经元，然后在最后预测时再把它们都加回来。而在稀疏数据上，这种范式可能就会不太合理。试想一个本身

信息就很少的数据，如果我们想用神经网络在其中挖掘出信息的话，可能也只有有限的神经元能够做到。而 Dropout 却有可能随机丢弃这些有限的神经元，所以就可能导致模型一直无法学习到稀疏特征背后的规律。

为此，笔者的做法是当数据的稀疏度达到 75% 以上时，就弃用 Dropout 这项技术：

```
01 def handle_sparsity(self):
02     if self.sparsity >= 0.75:
03         # 通过把 self.dropout_keep_prob 设置为 1
04         # 来启用 Dropout 这项技术
05         self.dropout_keep_prob = 1.
```

需要指出的是，这种做法没有相应的理论分析，更多的只是经验上的做法，所以在真正解决现实问题时，最好还是根据具体情况来修改 `self._handle_sparsity` 这个方法。

6.3 AutoBase 的实现补足

前两节我们把数据的准备与数据的转换这两大部分的各方面都大致叙述了一遍，它们其实已经构成了半自动化机器学习框架的核心部分。在本节中，我们会把剩余的一些工作给补足，并真正完成 AutoBase 的实现。

首先自然是 AutoBase 的初始化工作，从中也能看出不少数据准备、数据转换的影子，如代码 6.2 所示。

代码 6.2 AutoBase 的初始化：Base.py

```
01 class AutoBase:
02     """
03     初始化结构
04     name: 模型的名字
05     data_info: 数据超参数
06     pre_process_settings: 数据预处理器 PreProcessor 的参数设置
07     nan_handler_settings: 缺失值处理器 NanHandler 的参数设置
08     """
09     def init (self, name=None, data_info=None,
10              pre_process_settings=None, nan_handler_settings=None):
11         # 因为 AutoBase 有可能需要从文件格式的数据中读取数据
12         # 所以我们要求用户调用 AutoBase 时提供一个名字
13         # 从而方便我们找到相应的文件数据
14         if name is None:
15             raise ValueError("name should be provided when using AutoBase")
16         self.name = name
17
18         # 初始化数据的准备时所需要用到的属性
19         self.data_folder = None
20         self.whether_redundant = None
21         self.feature_sets = self.sparsity = self.class_prior = None
22         self.n_features = self.all_num_idx = self.transform_dicts = None
```

```
23
24     # 初始化数据超参数
25     if data_info is None:
26         data_info = {}
27     else:
28         assert msg = "data_info should be a dictionary"
29         assert isinstance(data_info, dict), assert msg
30     self.data_info = data_info
31     self.data_info_initialized = False
32     self.numerical_idx = self.categorical_columns = None
33
34     # 定义数据预处理器的相关属性
35     if pre_process_settings is None:
36         pre_process_settings = {}
37     else:
38         assert msg = "pre process settings should be a dictionary"
39         assert isinstance(pre_process_settings, dict), assert msg
40     self.pre_process_settings = pre_process_settings
41     self.pre_processors = None
42     self.pre_process_method = self.scale_method = None
43     self.reuse_mean_and_std = None
44
45     # 定义缺失值处理器的相关属性
46     if nan_handler_settings is None:
47         nan_handler_settings = {}
48     else:
49         assert msg = "nan handler settings should be a dictionary"
50         assert isinstance(nan_handler_settings, dict), assert msg
51     self.nan_handler_settings = nan_handler_settings
52     self.nan_handler = None
53     self.nan_handler_method = self.reuse_nan_handler_values = None
54
55     # 初始化数据预处理器的参数设置
56     self.init_pre_process_settings()
57     # 初始化缺失值处理器的参数设置
58     self.init_nan_handler_settings()
```

在完成了整个框架的初始化后，我们来看看如何对各个具体的参数进行初始化。首先来看看 AutoBase 中数据超参数的初始化：

```
59     def init_data_info(self):
60         # 如果已经初始化过数据超参数，就直接返回
61         if self.data_info_initialized:
62             return
63         # 将“已经初始化过的数据超参数”的属性置为 True
64         self.data_info_initialized = True
65         # 从 data_info 中获取 numerical_idx 等数据信息
66         # 如果没有相应信息，就将它们的值置为 None
67         # 从而就能应用之前章节的相应内容，完成自动的初始化
68         self.numerical_idx = self.data_info.get("numerical_idx", None)
```

```

69     self.categorical_columns = self.data_info.get(
70         "categorical_columns", None)
71     self.feature_sets = self.data_info.get("feature_sets", None)
72     # 数据的稀疏度和各类别的先验概率也进行类似的操作
73     self.sparsity = self.data_info.get("sparsity", None)
74     self.class_prior = self.data_info.get("class_prior", None)
75     # 如果 data_info 中提供了足够多的信息的话
76     # 就可以直接把相应的属性推算出来
77     if self.feature_sets is not None and self.numerical_idx is not None:
78         self.n_features = [len(feature_set)
79                             for feature_set in self.feature_sets]
80         self.gen_categorical_columns()
81     # 默认文件格式的数据存储在“Data”文件夹中
82     self.data_folder = self.data_info.get("data_folder", "Data")
83     # 默认文件后缀为txt，并随机打乱训练数据集
84     self.data_info.setdefault("file_type", "txt")
85     self.data_info.setdefault("shuffle", True)
86     # 默认交叉验证集占比10%
87     self.data_info.setdefault("test_rate", 0.1)
88     # 默认数据预处理阶段为第3阶段
89     self.data_info.setdefault("stage", 3)

```

可以看到，这些初始化在前文中都或多或少地有所提及。在完成数据超参数的初始化后，我们来看一下数据预处理器（PreProcessor）和缺失值处理器（NanHandler）这两个重要构件的参数的初始化。

```

90     def init_pre_process_settings(self):
91         # 默认使用标准化作为数据预处理的方法
92         # （事实上暂时也不支持其他方法，不过进行拓展是比较方便的）
93         self.pre_process_method = self.pre_process_settings.get(
94             "pre_process_method", "normalize")
95         # 默认使用截断法来处理异常值
96         self.scale_method = self.pre_process_settings.get(
97             "scale_method", "truncate")
98         # 默认在测试阶段不使用训练集的统计量
99         self.reuse_mean_and_std = self.pre_process_settings.get(
100             "reuse_mean_and_std", False)
101         # 初始化存储各个预处理器的字典 self.pre_processors
102         if self.pre_process_method is not None and self._pre_processors is None:
103             self.pre_processors = {}
104
105     def init_nan_handler_settings(self):
106         # 默认使用中位数来替代缺失值
107         self.nan_handler_method = self.nan_handler_settings.get(
108             "nan_handler_method", "median")
109         # 默认在测试阶段使用训练集的统计量来替代缺失值
110         self.reuse_nan_handler_values = self.nan_handler_settings.get(
111             "reuse_nan_handler_values", True)

```

有了这些初始化方法之后，结合前两节的各段代码，其实就已经涵盖 **AutoBase** 的所有主要功能了。不过为了尽善尽美，我们需要给用户提供一些针对文件格式数据集的接口，从而做到真正的、足够友好的“半自动化”。具体而言，我们至少需要实现如下两个功能。

- 从某个文件中完成数据读取与转换的工作，并返回一个模型能用的 **numpy** 数组。
- 将模型的预测标签转换为“真实的”标签。比如，假设一个数据集原始的标签为“是”和“否”，为了让神经网络进行相应的训练，我们需要在数据预处理阶段把这些原始标签“数值化”，比如把所有的“是”都转换为 1、把所有“否”都转换成 0。这样一来，我们的模型预测标签就会是 1 或 0。而用户在实际应用中，可能会想要我们预测“是”或“否”这两种标签，所以我们需要提供相应的标签转换方法。

由于我们之前的封装做得足够完善，所以这两个功能的实现比较简单。

```

112     # 定义一个能清除临时数据预处理器的方法
113     def _pop_preprocessor(self, name):
114         if isinstance(self._pre_processors, dict):
115             if name in self._pre_processors:
116                 self._pre_processors.pop(name)
117
118     def get_transformed_data_from_file(
119         self, file, file_type="txt", include_label=False
120     ):
121         # 利用相应的方法，从文件中获取原始数据
122         x, _ = self._get_data_from_file(file_type, 0, file)
123         # 利用相应方法，将原始数据转换成模型接收的 numpy 数组
124         # 其中，第二个参数“new”代指“这是一个新的数据集”
125         # 同时，它可能会创建一个名字为 new 的数据预处理器
126         x = self._transform_data(x, "new", include_label=include_label)
127         # 所以在得到想要的数据之后，我们要清除这个
128         # 有可能存在的、名字为 new 的临时数据预处理器
129         self._pop_preprocessor("new")
130         # 返回转换好的 x
131         return x
132
133     def predict_labels(self, x):
134         # 利用 get_labels_from_classes 方法，完成标签的转换工作
135         return self.get_labels_from_classes(self.predict_classes(x))

```

其中，`self.get_labels_from_classes` 方法的相关代码如下所示。

```

136     # 定义从原始标签到数值化标签的转换字典对应的属性
137     @property
138     def label2num_dict(self):
139         return None if not self.transform_dicts[-1] else self.transform_dicts[-1]
140
141     # 定义从数值化标签到原始标签的转换字典对应的属性
142     @property
143     def num2label_dict(self):
144         label2num_dict = self.label2num_dict

```

```

145     # 如果 label2num_dict 是 None, 就说明我们数值化过标签
146     # 这说明数值化标签和原始标签是一致的
147     if label2num_dict is None:
148         Return
149     # 否则, 就返回一个对应的 numpy 数组
150     # 由于 numpy 数组的索引功能很强大, 所以它能被当作“字典”来使用
151     num_label_list = sorted([(i, c) for c, i in label2num_dict.items()])
152     return np.array([label for _, label in num_label_list])
153
154     def get_labels_from_classes(self, classes):
155         # 获取转换字典
156         num2label_dict = self.num2label_dict
157         # 根据字典来获取原始标签
158         if num2label_dict is None:
159             return classes
160         return num2label_dict[classes]

```

以上,我们就完成了 AutoBase 的所有实现。不过为了真正应用它,我们还需要使用到 Python 中的“黑魔法”之一——元类 (Meta Class)。

6.4 AutoMeta 的实现

前三节我们把 AutoBase 的所有代码实现都介绍了一遍,但是还没有介绍怎样具体地将它应用到已有的、基于 TensorFlow 基本模型框架 (Base) 拓展的机器学习模型 (model) 上 (比如 BasicNN 或 AdvancedNN 等)。在本章的开头我们曾经说过, AutoBase 是一个非常通用的技术,能够作用于几乎任意一个机器学习模型上。换句话说,它应当可以无缝地融进已有的 model 中,并提供前三节所提到过的所有功能。但是,此前实现的 AutoBase 只完成了它自身功能的实现,如果想让它融进 model 的话,我们还需要实现一个相应的沟通 AutoBase 与 model 这两者的桥梁。为此,我们就需要实现 AutoMeta, 并引入“元类 (Meta Class)”这个 Python 中非常神奇、非常强大的概念。

本节将介绍的元类充分体现了 Python 中的一句名言:万物皆对象。具体来说,附录 C 中将会介绍的“装饰器 (Decorator)”告诉了我们“函数 (Function) 亦对象”,而元类则会告诉我们“类 (Class) 亦对象”。

注意: 元类和附录 C 中介绍的装饰器在思想上有许多相通之处,而装饰器也是 Python 中非常重要的概念,所以笔者个人建议大家可以先行翻阅附录 C 来对“万物皆对象”这个思想有更好的理解。

所谓的“类亦对象”,其实和附录 C 中“函数亦对象”的思想类似:它意味着类可以被赋值给变量、通过变量也能创建该类的实例。举一个例子:

```

01 class Class:
02     def __init__(self):
03         self.x = 1

```

```
04
05 one = Class
06 print(one().x)
```

上述这段代码将会输出 1。

正如装饰器返回的是一个函数，我们可以认为元类返回的是一个类。也正如我在讲装饰器时说过的，装饰器的核心思想，其实就是装饰函数这个对象，然后让函数在自身代码不变的情况下增添一些具有普适性的功能。在我看来，元类的核心思想，就是摆弄类这个对象，使得我们能够对其有最高程度的控制权。

需要指出的是，这不一定是准确的理解！正如 Python 界的领袖 Tim Peters 说过：“元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要什么做什么，而且根本不需要解释为什么要用元类。”

因此笔者的个人理解仅仅来自笔者对元类（在半自动化机器学习框架中）的应用，它很有可能是非常片面的。如果大家有自己的想法的话，完全不必被这种理解限制住思维，它仅仅能起到一个（还算不错的）参考的作用。

那么什么叫作最高程度的控制权呢？一个比较简单的例子就是实现如下需求：

- 定义一个“人”（Person）类，它有三个方法：吃饭、睡觉、学 Python。
- 定义 Person 的三个子类“小李（Li）”“小张（Zhang）”“小王（Wang）”。
- 定义 Person 的另一个子类“小红（Hong）”，要求她：
 - 吃饭像小李一样快。
 - 睡觉像小张一样香。
 - 学 Python 像小王一样勤快。

那么此时应该怎么去实现呢？如果再要求把上面三个要求换一换顺序呢？

也许 Python 中会有许多其他的解决方案，但是笔者所知道的最简单的方法，就是使用元类来实现。幸运的是，虽然元类的思想可能很深，但就这个简单的问题而言，即使我们不进行任何说明，相信聪明的读者也能读懂下面这几段代码。

先定义 Person 类：

```
01 class Person:
02     def init (self):
03         self.ability = 1
04
05     def eat(self):
06         print("Eat: ", self.ability)
07
08     def sleep(self):
09         print("Sleep: ", self.ability)
10
```



```

11     def learn python(self):
12         print("Learn Python: ", self.ability)

```

再定义三个子类：

```

13 class Li(Person):
14     def eat(self):
15         print("Eat: ", self.ability * 2)
16
17 class Zhang(Person):
18     def sleep(self):
19         print("Sleep: ", self.ability * 2)
20
21 class Wang(Person):
22     def learn python(self):
23         print("Learn Python: ", self.ability * 2)

```

然后是最关键的定义元类（Meta Class）：

```

24 class Mixture(type):
25     def __new__(mcs, *args, **kwargs):
26         name, bases, attr = args[:3]
27         person1, person2, person3 = bases
28
29         def eat(self):
30             person1.eat(self)
31
32         def sleep(self):
33             person2.sleep(self)
34
35         def learn python(self):
36             person3.learn python(self)
37
38         attr["eat"] = eat
39         attr["sleep"] = sleep
40         attr["save life"] = save life
41
42         return type(name, bases, attr)

```

这样就完成了所有的实现。不过，读者可能发现其中有三行代码显得“特别傻”——没错，确实可以用更具有普适性的三行代码来代替上述代码的第 38~40 行：

```

for key, value in locals().items():
    if str(value).find("function") >= 0:
        attr[key] = value

```

抛开所有技术细节而只谈应用的话，其实上面这个例子可能已经足够了，接下来就让我们测试一下这个 Mixture 元类。先来定义一个小的测试函数，它依次调用 Person 实例中的“吃饭”“睡觉”“学 Python”这三个动作：

```

01 def test(person):

```

```

02     person.eat()
03     person.sleep()
04     person.learn_python()

```

然后进行两组测试：

```

01 # 依次获得小李的吃饭能力、小张的睡觉能力和小王的学 Python 能力
02 class Hong(Li, Zhang, Wang, metaclass=Mixture):
03     pass
04
05 # 将会输出:
06 # Eat: 2
07 # Sleep: 2
08 # Learn Python: 2
09 test(Hong())
10
11 # 依次获得小张的吃饭能力、小李的睡觉能力和小王的学 Python 能力
12 class Hong(Zhang, Li, Wang, metaclass=Mixture):
13     pass
14
15 # 将会输出:
16 # Eat: 1
17 # Sleep: 1
18 # Learn Python: 2
19 test(Hong())

```

可以看到，我们确实获得了类的高度控制权。虽然这个例子很简单，不过其实即使仅仅基于该例子的思想，就已经可以弄出许多有意思的应用了。事实上我们接下来马上就要介绍的、辅助 AutoBase 与 model 进行融合的 AutoMeta，就是用这个思想来实现的。首先来看看它的初始化，它自然应该包括 AutoBase 的初始化和 model 的初始化这两部分。同时，由于 AutoBase 的主要功能在于自动化数据预处理的部分，结合先前实现各种 model 时做的“输入数据都已经过数据预处理”这个假设，不难想象 AutoBase 的初始化应该放在 model 的初始化的前面，如代码 6.3 所示。

代码 6.3 AutoMeta 的初始化：Base.py

```

01 class AutoMeta(type):
02     def new (mcs, *args, **kwargs):
03         name, bases, attr = args[:3]
04         # 用 auto base 变量代指 AutoBase, 用 model 变量代指 model
05         # 这里要特别注意模型继承的顺序: 先 AutoBase, 再 model
06         auto_base, model = bases
07
08         def init (self, name=None, data_info=None,
09                   model_param_settings=None, model_structure_settings=None,
10                   pre_process_settings=None, nan_handler_settings=None
11                 ):
12             # 先调用 AutoBase 的初始化方法
13             auto_base.__init__(self, name, data_info,

```

```

14         pre process settings, nan handler settings)
15     # 根据 model 的签名来进行不同的初始化
16     if model.signature != "Advanced":
17         # 如果 model 的签名不是 Advanced (比如是 BasicNN)
18         # 就说明不需要用数据超参数来初始化模型
19         model. init (self, name,
20                     model param settings, model structure settings)
21     else:
22         # 否则, 就需要用数据超参数来完成 model 的初始化
23         model. init (self, name, data info,
24                     model param settings, model structure settings)
25
26     # 由于数据方面的操作都是由 AutoBase 来维护的
27     # 所以数据超参数的初始化就只需调用 AutoBase 的相应初始化即可
28     def init_data_info(self):
29         auto_base.init_data_info(self)

```

在完成了初始化之后, 我们还需要关注模型的保存、模型的训练与预测, 以及一些与 6.3 节中类似的、针对文件格式的数据的友好的接口。

首先来看看模型的保存。由第 3 章和第 5 章的讨论可知, 我们只需维护模型保存时需要保存的属性即可。对于 **AutoBase** 而言, 由于它只引入了数据预处理的相应逻辑, 所以我们也只需要将数据预处理相应的属性进行保存。

```

30     def _define_py_collections(self):
31         # 先调用 model 的相应方法
32         model._define_py_collections(self)
33         # 然后在 self.py_collections 这个记录将要保存的属性的列表中
34         # 加入各种数据预处理相关的属性
35         self.py_collections += [
36             "pre_process_settings", "nan_handler_settings",
37             "pre_processors", "nan_handler", "transform_dicts",
38             "numerical_idx", "categorical_columns", "transform_dicts"
39         ]

```

然后是模型的训练与预测。其中模型的训练尤为简单, 因为 **AutoBase** 本身不干预训练, 所以 **AutoMeta** 只需直接调用 **model** 的训练方法即可。至于模型的预测的话, 由于之前相应的封装做得很好, 所以其实现和上一节说过的 `self.get_transformed_data_from_file` 方法的实现类似, 都是比较简单的。

```

40     def fit(self, x=None, y=None, x_test=None, y_test=None,
41            sample_weights=None, names=("train", "test"),
42            timeit=True, snapshot_ratio=3, print_settings=True, verbose=1
43    ):
44         # 直接调用 model 的 fit 方法, 完成训练
45         return model.fit(
46             self, x, y, x_test, y_test, sample_weights, names,
47             timeit, snapshot_ratio, print_settings, verbose
48         )

```

```

49
50     def predict(self, x):
51         # 利用 self.transform data 方法, 对原始数据 x 进行转换
52         x = self.transform data(x, "new", include label=False)
53         # 利用转换好的 x 进行相应的概率预测
54         rs = self.predict(x)
55         # 清除可能存在的临时数据预处理器
56         self.pop preprocessor("new")
57         # 返回预测结果
58         return rs
59
60     def predict classes(self, x):
61         # 如果是回归问题, 则不允许调用“预测类别”这个方法
62         if self.n class == 1:
63             raise ValueError(
64                 "Predicting classes is not permitted in regression problem")
65         # 利用上述重新实现的 predict 方法, 完成类别的预测
66         return self.predict(x).argmax(1).astype(np.int32)

```

最后就是提供一些针对文件格式的数据的接口了。由于我们在上一节实现了能从某个文件中完成所有数据准备工作的 `self.get_transformed_data_from_file` 方法, 所以在此基础上进行各种拓展是比较直观的:

```

67     # 定义一个能直接从文件中的数据获取模型概率预测的方法
68     def predict from file(self, file, file type="txt", include label=False):
69         # 利用相应方法, 从文件中获取准备好的数据
70         x = self.get transformed data from file(
71             file, file type, include label)
72         # 如果文件中含有标签列的话, 就去掉最后一列
73         if include label:
74             x = x[:, :-1]
75         # 调用相应方法, 完成模型的概率预测
76         return self.predict(x)
77
78     # 定义一个能直接从文件数据获取模型类别预测的方法
79     # 由于实现的方式与上一个方法以及 predict classes 方法大同小异
80     # 所以相应注释从略
81     def predict classes from file(
82         self, file, file type="txt", include label=False
83     ):
84         if self.numerical idx[-1]:
85             raise ValueError(
86                 "Predicting classes is not permitted in regression problem")
87         x = self.get transformed data from file(file, file type, include label)
88         if include label:
89             x = x[:, :-1]
90         return self.predict(x).argmax(1).astype(np.int32)
91
92     # 定义一个能直接从文件数据获取模型标签预测的方法

```

```

93     def predict_labels_from_file(self, file, file_type="txt", include_label=False):
94         # 先调用相应方法获取类别预测
95         classes = self.predict_classes_from_file(file, file_type, include_label)
96         # 然后调用相应方法把类别预测转换为标签预测
97         return self.get_labels_from_classes(classes)

```

至此，AutoMeta 的核心功能全部实现完毕，接下来要做的就只有收尾工作了。

```

98     for key, value in locals().items():
99         if str(value).find("function") >= 0:
100             attr[key] = value
101     return type(name_, bases, attr)

```

可以看到，这里这 4 行代码和本节一开始所举的例子中的最后 4 行代码是完全一致的。

注意：笔者在实现 AutoMeta 时，还额外地实现了两个功能：一个是第 4 章提到过的 `evaluate` 方法的重写，另一个则是 `predict_target_prob` 方法的实现，它能输出模型对某个原始标签的概率预测。之所以没有对它们进行展示，主要是希望大家能够通过之前的内容尝试自己推导出相应的实现方式，笔者的实现可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/Base.py。

现在我们既有半自动化机器学习框架 AutoBase，又有沟通 AutoBase 与 model 之间的桥梁 AutoMeta 了，那么究竟如何将这些东西套用在神经网络上面呢？得益于元类的强大，相应的实现代码是目前为止见过的代码中最为简洁的：

```

01 # 导入之前实现的增强版神经网络 AdvancedNN
02 from Dist.NeuralNetworks.e_AdvancedNN.NN import Advanced
03 # 导入前四节实现的 AutoBase 和 AutoMeta
04 from Dist.NeuralNetworks.Base import AutoBase, AutoMeta
05
06 # 实现 AutoAdvanced
07 # 注意一定要把 AutoBase 放在第一位，把 Advanced 放在第二位
08 class AutoAdvanced(AutoBase, Advanced, metaclass=AutoMeta):
09     pass

```

可以看到，只需短短的两行代码，我们就把 AutoAdvanced 实现完毕了。这是因为我们已经把主要的功能实现都放在了 AutoBase 中，并把主要的和 model 相关联的逻辑放在了 AutoMeta 中，所以对于 AutoAdvanced 本身而言无须进行任何进一步的实现。

值得一提的是，由于 AutoBase 和 AutoMeta 能够应用于任何 model，所以我们完全可以将半自动化机器学习框架应用在朴素神经网络（BasicNN）上：

```

10 from _Dist.NeuralNetworks.c_BasicNN.NN import Basic
11
12 class AutoBasic(AutoBase, Basic, metaclass=AutoMeta):
13     pass

```

不过由于 AdvancedNN 通常都会比 BasicNN 的性能要好，而且 AdvancedNN 从功能上来说

是包含 BasicNN 的，所以一般我们只会用到 AutoAdvanced 而不会用到 AutoBasic。不过至少从 AutoBasic 的实现可以看出，AutoBase 确实能够套用在几乎任意一个 model 上（我们在附录 A 中实现 LinearSVM 与 SVM 时还会体现出这一点）。

至此，半自动化机器学习框架的所有相关实现就全部做完了。对于其具体的应用方式及实际效果，我们会在第 7 章介绍完工程化机器学习框架（DistMixin）之后进行统一介绍与展示，这里就暂时先按下不表。

6.5 训练过程的监控

本章的前四节我们叙述了半自动机器学习框架的所有功能以及相应的实现，这一节我们将对在第 3 章就已经出现过的但却一直没有展开介绍的非常重要的东西——训练监控器（TrainMonitor）进行补充说明。毫不夸张地说，虽然我们在 TensorFlow 模型的基本框架（Base）中已经应用了 TrainMonitor，但是它却是 AutoBase 之所以敢自称“半自动化”的重要原因之一。

顾名思义，TrainMonitor 的作用就是监控机器学习模型 model 的训练过程，其两大具体行为在 3.5.3 节的代码 3.3 中也有所提及：

- 若发现 model 出现过拟合现象，则提前终止训练。
- 若发现 model 存在欠拟合现象，则延长训练步数。

也正如第 3 章所说，TrainMonitor 是非常重要的工具，它能让我们免于调整训练步长，从而把注意力集中在 model 本身的调优上。但是，虽然上述两种行为叙述起来浅显易懂，然而真正实现起来却有诸多需要解决的问题，因此我们单独设置了一整节的内容以便从监控原理、实现思路与具体代码这三大方面来详细地阐述它。

6.5.1 监控训练过程的原理

为了实现 TrainMonitor 的两大具体行为，我们需要先将某些概念翻译成机器能够理解的公式。首先是“出现过拟合现象”这个概念，由过拟合本身的定义可知，我们可以用交叉验证（Cross Validation，后简称 CV）集上 model 的表现来进行相应的判断：当 CV 集上的效果变差时，我们就可以认为 model 出现了过拟合。

但是这个“变差”的定义方式却有很多。一种朴素的做法是：一旦发现 model 在 CV 集上的表现在连续 n 次的迭代中都越来越差，那么就认为它出现了过拟合。然而，这种做法却可能会面临这样两个问题：

- 当 n 较小时，如果数据集较难的话，由于 model 的训练较慢，所以可能每次迭代中 CV 上的表现的变化都很小，此时 model 的表现就很有可能在连续的 n 个迭代中每次差一点点。然而即使每次变差的程度可以忽略不计，上述的朴素做法仍然会认为出现了过拟合。

- 当 n 较大时，就很有可能监测不到过拟合现象了。因为即使 model 已经陷入过拟合，它在 CV 集上的表现有时还是能“回光返照”一下的。因此如果 n 较大的话，这种回光返照就会打断表现变差的连续性，从而导致一直无法出现“连续 n 个迭代中表现越来越差”的现象。

总结一下，这种朴素的做法太过死板。它无法区分开变差程度的大小所应该带来的判别上的差异，也无法识别出连续变差的过程中那短暂的变好只是偶然因素。为了解决这两个问题，一种自然的想法就是：我们不按照“变差了多少次”来判断 model 是否过拟合，而是按照“变差了多少”来进行判断。具体而言，假设我们以准确率（acc）作为 model 好坏的指标的话，那么如果 model 在 CV 集上的 acc 累计变差了 0.1（比如，从 0.95 一直变到了 0.85），就认为 model 出现了过拟合现象。

不难发现这种做法可以写得更一般化，即其本质其实是通过监控“过拟合程度” R_t 与提前设定好的阈值 M_R 之间的关系来判断 model 是否过拟合（overfitted）：

$$R_t \geq M_R \Rightarrow \text{overfitted}$$

只不过，这种做法是将上述 R_t 设置为了“累计变差了多少”而已。这种做法从直观上来说比第一种朴素的做法而言要合理很多：当 model 每次都只变差一点点时，即使是连续变差了十几次迭代，也不会被认为 model 陷入了过拟合；同时如果 model 在不断地变差，而只是偶尔变好一下的话，这种做法也能将变差的程度很好地累计下来，从而受那“回光返照”的影响就不会太大。但是，如果我们仔细推敲这种做法的话，就会发现它还是有可能出现致命的问题的。事实上，这种做法的一种最自然的实现方式是：通过累计本次迭代与上次迭代的 model 表现之间的差异来判断 model 是否出现过拟合现象。在这种实现下，如果假设 model 在第 t 次迭代中的表现为 r_t 的话，那么过拟合程度 R_t 就能写成

$$R_t = \sum_{i=1}^t (r_i - r_{i-1})$$

不难看出，上式可以化为更加简单的

$$R_t = r_1 - r_0 + r_2 - r_1 + \cdots + r_t - r_{t-1} = r_t - r_0$$

也就是说，如果累计的差异是“本次迭代与上次迭代之间的差异”的话，那么此时过拟合程度其实就只是用第 t 次迭代中的表现 r_t 与 model 的初始表现 r_0 之间的差异来衡量，这当然是极其不合理的做法。因为我们知道，即使是 model 陷入了严重的过拟合，其表现通常仍然会比初始化时的表现要好不少。

所以我们就需要对累计的差异进行更合理的设计。从直观上来说，由于基于梯度下降法的 model（比如神经网络）在训练初期的性能提升会非常快，而到后期则会越来越慢直至最后收敛，所以上述差异累计的方式的不合理之处就在于，它没有区别对待训练初期阶段的差异与训练后期阶段的差异。针对该不合理之处，解决方案就呼之欲出了：我们的差异不应从头开始累计，而应该只累计“近期的”差异。表述成数学语言的话，假设我们只累计最近的 k 个差异的话，

过拟合程度 R_t 就能写成

$$R_t = \sum_{i=t-k+1}^t (r_i - r_{i-1}) = r_t - r_{t-k}$$

这样一来就显得合理不少，但是当样本量不足时，仍然存在着两个问题：

- 当 k 设置得较小时，由于样本量不足时 model 表现的波动很大，所以可能会出现 model 表现突然变差很多的情况，此时我们就有可能过早地停止训练。
- 当 k 设置得较大时，由于样本量不足时 model 训练得很快，所以 model 往往会很快陷入过拟合，此时我们就有可能会过晚地停止训练。

仔细分析这两个问题不难发现，它们背后的诱因是一致的：那就是每次计算当前差异时，都是计算当前 model 表现与上一次的 model 表现的差异，这使得累计的差异在 model 波动较大时非常不稳定。为此，我们就需要引入更合理的计算差异的方式。相关的、切实有效的解决方案有不少，这里介绍一种笔者发明并经常采用的，也是我们后文将会进行具体代码实现介绍的手段。

- 计算当前差异时，计算当前表现与最近 k 次迭代的平均表现（一般称之为窗口长度为 k 的 running mean，可翻译为“滑动均值”）的差异。
- 计算过拟合程度时，累计的不是单纯的、当前的差异，而是当前差异与最近 k 次迭代的表现的标准差（一般称之为窗口长度为 k 的 running std，可翻译为“滑动标准差”）之间的比例。

用数学语言来表达的话，就是

$$R_t = \sum_{i=t-k+1}^t \frac{r_i - \mu_i}{\sigma_i}$$

其中

$$\mu_i = \frac{1}{k} \cdot \sum_{j=i-k+1}^i r_j$$

$$\sigma_i = \sqrt{\frac{1}{k-1} \cdot \sum_{j=i-k+1}^i (r_j - \mu_j)^2}$$

这样一来，如果 model 表现的波动很大，那么一方面当前差异 ($r_t - \mu_t$) 会没那么大（因为比较的是 running mean），另一方面则由于此时 running std 会很大，所以过拟合程度受当前差异的影响也会变小。反之，如果 model 表现的波动很小的话，过拟合程度受当前差异的影响就会变大。总而言之，此时过拟合程度受当前差异影响的大小与 model 表现波动的大小呈负相关，这是符合直观的，同时也确实是在实际任务中之行之有效的。

有时为了让这种定义下的过拟合程度对极端情况更加稳健，会采用如下的累计方式：

$$R_t = \sum_{i=t-k+1}^t \min\left(\frac{M_R}{3}, \frac{r_i - \mu_i}{\sigma_i}\right)$$

换句话说，我们至少需要累计三次差异才能判定 model 陷入了过拟合。

不过需要指出的是，虽然这种过拟合程度的定义确实已经比较可靠，但我们总是能够构造出一些极端的情况，使得这种过拟合程度仍然无法反映出真实的过拟合程度。我们前文大部分时间在做的，其实只是降低这种情况出现的概率。也正因此，我们应该不断地对 model 当前的参数做一个备份（checkpoint），并在 model 陷入过拟合之后，利用该备份将 model 的参数还原为过拟合之前的参数。这里就涉及两个问题：

- 做备份的条件是什么？
- 做备份的频率是多少？

对这两个问题的解答将会把 TrainMonitor 引导至两个不同的方向。一个是追求训练速度，适当降低对性能的要求，此时做备份的条件将会设定得较严格，做备份的频率也会调低；另一个就是追求 model 的性能，不太在乎时间的开销，此时做备份的条件将会放宽，做备份的频率也会调高。所以究竟应该如何解答这两个问题需要视情况而定，笔者在此给出一种中庸的、普遍适用的解决方案以供参考。

- 周期性地备份最佳 model 参数。
- 此外，在出现如下三种特殊情形时备份 model 的参数。
 - 刚陷入过拟合，且当前表现为历史最优时。
 - 刚从过拟合状态恢复且当前表现与历史最优之间只差一个 running std 之内时。
 - 当需要备份最优参数，且当前表现为历史最优时。其中，需要备份最优参数的情形包括如下两种。
 - i. 陷入过拟合之后。
 - ii. model 性能出现飞跃性进步时。

以上就是对如何判断 model 是否陷入过拟合的讨论。注意，在本节的一开始我们曾经说过，TrainMonitor 除了需要在 model 过拟合时终止训练，还需要在发现 model 存在欠拟合时延长训练步数。一个很自然的想法就是，如果我们认为 model 没有陷入过拟合（即过拟合程度 R_t 没有超过阈值 M_R ），那么就认为 model 存在欠拟合现象。不过在这种情况下，我们需要给出一些额外的终止条件，否则如果 model 收敛得很好的话，那么它永远也不会陷入过拟合，此时我们的训练就会一直持续下去。

同样的，额外的终止条件会有很多种提法，这里就提供其中两个直观且有效的条件。

- 设置一个“最大训练步数（max_epoch）”，如果训练步数（epoch）超过它的话就强行终止训练。
- 如果 model 在较长的时间内都没有性能提升的话，就直接终止训练。

其中第一个条件是很好实现的，第二个条件还需要花一些功夫来把它转化为数学的语言。

具体来说, 如果 model 表现的 running std 比某个阈值 (m_r) 还小, 由于这意味着此时 model 的性能趋于稳定, 即 model 很有可能已经开始收敛, 那么我们就开始对 model 的“稳定程度”正向计数。反之如果 running std 比阈值 m_r 要大, 我们就认为 model 脱出了收敛的空间, 开始了新一轮的探索, 所以我们就开始对 model 的“稳定程度”反向计数。这样一来, 只要稳定程度的计数累积到了某个程度, 我们就认为 model 已经稳定了下来, 无法再获得性能提升, 从而就提前终止训练。

此外, 从直观上来说, 延长训练步数不应该是“无偿”的, 我们应该对延长训练步数这种行为做出一些惩罚。比如, 每延长一次步数就给过拟合程度 R_t 加一个数, 这样也能在一定程度上防止 model 不断地延长训练步数。

6.5.2 监控训练的实现思路

由上一节的讨论可知, 模型的过拟合程度是用模型当前的表现与 running mean、running std 之间的累计差异来衡量的。换句话说, 我们需要不断地计算 running mean (后简称 rm) 与 running std (后简称 rs)。虽然 numpy 中已经有 mean 和 std 相应的计算函数, 但是对于这种滑动窗口下的计算而言, 直接简单粗暴地调用这些函数来计算反而可能会让效率变低。以 rm 为例, 假设现在一共有 n 个数据 ($r_1 \sim r_n$), 而滑动窗口的长度为 m , 那么不难看出此时我们需要计算 $n - m + 1$ 个 rm:

$$rm_t = \frac{1}{m} \sum_{i=t}^{t+m-1} r_i$$

如果直接调用 numpy 中的函数来计算的话, 就需要进行 $(n - m + 1) \cdot m$ 次加法和 m 次除法。但是如果仔细想想的话, 不难发现各个 rm 之间并非是毫无关联的。事实上, 只需令

$$\widetilde{rm}_t \triangleq m \cdot rm_t$$

就有

$$\widetilde{rm}_t = \sum_{i=t}^{t+m-1} r_i = r_t + \sum_{i=t+1}^{t+m} r_i - r_{t+m} = r_t + \widetilde{rm}_{t+1} - r_{t+m}$$

换句话说, 只要有了 \widetilde{rm}_t , 那么只需要再做一个加法和一个减法就能得到 \widetilde{rm}_{t+1} , 而如果直接计算 \widetilde{rm}_{t+1} 的话, 不难看出是需要做 m 个加法的, 这里直接就将计算量节省到原来的 0.5m。

而对于 rs 而言, 考虑到

$$rs_t = \sqrt{\frac{1}{m-1} \sum_{i=t}^{t+m-1} (r_i - rm_t)^2}$$

乍一看好像难以像 rm 那样进行递推, 不过由于

$$\sum_{i=t}^{t+m-1} (r_i - rm_t)^2 = \sum_{i=t}^{t+m-1} r_i^2 - 2 \cdot \sum_{i=t}^{t+m-1} r_i \cdot rm_t + \sum_{i=t}^{t+m-1} rm_t^2$$

且

$$rm_t = \frac{1}{m} \sum_{i=t}^{t+m-1} r_i$$

从而就有

$$\sum_{i=t}^{t+m-1} (r_i - rm_t)^2 = \sum_{i=t}^{t+m-1} r_i^2 - \frac{1}{m} \left(\sum_{i=t}^{t+m-1} r_i \right)^2 = \sum_{i=t}^{t+m-1} r_i^2 - m \cdot rm_t^2$$

那么，只需令

$$\tilde{rs}_t \triangleq (m-1) \cdot rs_t^2 + m \cdot rm_t^2$$

就有

$$\tilde{rs}_t = r_t^2 + \tilde{rs}_{t+1} - r_{t+m}^2$$

综上所述，只要我们把初始的 rm 和 rs 计算出来，那么通过

$$\tilde{rm}_t = r_t + \tilde{rm}_{t+1} - r_{t+m}, \quad \tilde{rs}_t = r_t^2 + \tilde{rs}_{t+1} - r_{t+m}^2$$

这两个递推公式，就能利用当前已有的 rm 和 rs 来快速地计算出最新的 rm 与 rs ，这将会是我们进行具体实现时要做的第一件事。

而我们要做的第二件事，就是返回值的管理。由于我们把 `TrainMonitor` 单独地抽象了出来，所以它必定要实现这样一个方法，该方法接收的参数为模型当前的表现，然后需要告诉调用者该模型是否陷入了过拟合、是否需要进行模型参数的备份等。这样一来，该方法的返回值就需要包含非常多的信息。

一个简单易懂的方法是，把所有信息依次返回，然后调用者按照顺序依次读入这些信息。这样做虽然不能说有错，但是其可读性、可拓展性和可维护性都比较差，不能算是“好的”实现。取而代之的，在这种需要返回很多信息的场景，我们一般会返回一个字典，然后该字典储存着所有需要用到的信息。在这种实现下，可读性能够通过该字典的键值（key）来保证，且如果我们想要额外地返回一些信息的话，只需往该字典中加入新的键-值对（key-value pair），就能轻松地完成拓展。事实上在具体的代码实现中，我们会用一个属性（`self.info`）来记录这个储存信息的字典，这样当需要获得模型是否陷入过拟合的相关信息时，只需调用该属性即可。

6.5.3 监控训练的具体代码

本节我们将会展示前两节的内容所对应的代码。虽然这些代码基本没有涉及算法层面上的东西（除了 rm 和 rs 的维护用到了点点算法），但是里面的逻辑较为复杂，所以笔者将原来作为一个整体的核心方法（`check`）拆分成若干个小的环节以便大家理解，核心方法本身则成为串联各个部分的中轴线。

首先来看看 `TrainMonitor` 的初始化，如代码 6.4 所示。

代码 6.4 训练监控器的实现：NNUtil.py

```

01 class TrainMonitor:
02     """
03     初始化结构
04     sign: 模型所用的评价指标 (metric) 的“符号”
05         当 sign 为 1 时意味着 metric 越大越好 (比如准确率 acc)
06         当 sign 为 -1 时意味着 metric 越小越好 (比如均方误差 mse)
07     snapshot_ratio: “周期性检查是否需要进行参数备份”中的“周期”
08     history_ratio: 滑动窗口长度相对于上述“周期”的比率
09     tolerance_ratio: 阈值  $M_R$  相对于上述“周期”的比率
10     extension: 每次延长的训练步数
11     std_floor、std_ceiling: 阈值  $m_r$  和提高鲁棒性的参数
12     """
13     def __init__(self, sign, snapshot_ratio, history_ratio=3, tolerance_ratio=2,
14                 extension=5, std_floor=0.001, std_ceiling=0.01):
15         self.sign = sign
16         self.snapshot_ratio = snapshot_ratio
17         # 记录滑动窗口长度的属性
18         self.n_history = int(snapshot_ratio * history_ratio)
19         # 记录阈值  $M_R$  的属性
20         self.n_tolerance = int(snapshot_ratio * tolerance_ratio)
21         self.extension = extension
22         self.std_floor, self.std_ceiling = std_floor, std_ceiling
23         # 记录 model 所有历史“分数”的属性
24         self._scores = []
25         # 记录是否开始对稳定程度计数的属性
26         self.flat_flag = False
27         # 记录 model 历史最优表现的属性
28         self._running_best = None
29         # 记录 model 当前表现是否是历史最优表现的属性
30         self._is_best = None
31         # 记录“滑动求和”的属性
32         self._running_sum = None
33         # 记录“滑动平方和”的属性
34         self._running_square_sum = None
35
36         # 记录 model 刚陷入过拟合时的表现的属性
37         self._over_fit_performance = math.inf
38         # 记录 model 最佳 checkpoint 的表现的属性
39         self._best_checkpoint_performance = -math.inf
40         # 记录“过拟合程度”  $R_t$  的属性
41         self._descend_counter = 0
42         # 记录“稳定程度计数”的属性
43         self._flat_counter = 0
44         # 记录 model 是否正处于过拟合状态的属性
45         self.over_fitting_flag = 0
46         # 记录 model 过拟合信息的字典, 该字典的 key 的具体意义如下:
47         # 1) terminate: 是否需要终止训练
48         # 2) save_checkpoint: 是否需要备份当前参数

```

```

49     # 3) save best: 是否需要备份最优参数
50     # 4) info: 想要告诉调用者的额外信息
51     self.info = {
52         "terminate": False, "save_checkpoint": False,
53         "save best": False, "info": None
54     }
55
56     # 记录延长步数所带来的惩罚的属性
57     self._descend_increment = self.n_history * extension / 30
58
59     def punish_extension(self):
60         # 每次延长步数时, 就给 $R_t$ 加一个数
61         self._descend_counter += self._descend_increment

```

在完成了初始化之后, 其实就只需再把核心方法 `check` 实现出来即可。正如前文所说, 该方法需要接收 `model` 的最新表现, 然后根据这个最新表现来判断 `model` 的过拟合程度, 并把相应的信息返回给调用者:

```

62     # 参数 new metric 即为 model 最新的表现
63     def check(self, new_metric):
64         # 根据 self.sign 属性, 获取 model 最新表现的“分数 (score)”
65         # 需要指出的是, 不管 model 采用的 metric 是越大越好 (如 acc)
66         # 还是越小越好 (如 mse), 由于这里乘了一个 self.sign
67         # 所以得到的 score 总是越大越好的
68         last_score = new_metric * self.sign
69         self._scores.append(last_score)
70         # 算出实际的滑动窗口长度
71         n_history = min(self.n_history, len(self._scores))
72         # 如果实际滑动窗口长度为 1, 就无须进行任何判断
73         # 因为此时模型才刚开始训练
74         if n_history == 1:
75             return self.info
76         # 调用 self._update_running_info 方法
77         # 从而使得在更新 rm、rs 的同时
78         # 获得当前模型表现 (相比于历史最优表现) 的提升程度
79         improvement = self._update_running_info(last_score, n_history)
80         # 在一开始把“是否备份参数”的 value 初始化为 False
81         self.info["save_checkpoint"] = False
82         # 由于 self._running_sum 记录的是“滑动求和”
83         # 所以滑动均值 (rm) 就是它除以实际滑动窗口的长度
84         mean = self._running_sum / n_history
85         # 根据公式计算出滑动标准差 (rs)
86         std = math.sqrt(max(
87             self._running_square_sum / n_history - mean ** 2, 1e-12))
88         # 规定 std 不能超过 self.std_ceiling 这个提高鲁棒性的属性
89         # 因为从经验上来看, 当滑动标准差 (rs) 太大时
90         # 一般都是异常的情况, 应该予以遏制
91         std = min(std, self.std_ceiling)
92         # 如果 rs 比阈值 $m_r$ 要小, 而且已经开始对稳定程度计数的话

```

```

93     # 就给记录稳定程度的 self. flat counter 加上 1
94     if std < self.std floor:
95         if self.flat flag:
96             self. flat counter += 1
97     # 否则
98     else:
99         # 1) 给记录稳定程度的 self. flat counter 减 1
100        self. flat counter = max(self. flat counter - 1, 0)
101        # 2) 算出最新 score 和 rm 之间的差 res
102        res = last score - mean
103        # 如果 res 小于负的 rs
104        # 而且最新 score 比刚陷入过拟合时的 score 也要差一个 rs 的话
105        if res < -std and last score < self. over fit performance - std:
106            # 我们就认为 model 正陷入过拟合
107            # 所以就调用相应的方法来处理
108            self. handle overfitting(last score, res, std)
109        # 而如果 res 比 rs 大的话
110        elif res > std:
111            # 我们就认为 model 可能正在从过拟合状态中恢复
112            # 所以同样调用相应的方法来处理
113            self. handle recovering(improvement, last score, res, std)
114        # 接下来就是根据上面更新过后的过拟合程度的信息来进行收尾的阶段
115        # 1) 如果稳定程度计数超过了阈值
116        if self. flat counter >= self.n tolerance * self.n history:
117            # 就告诉调用者 “model 的性能已经没有提升了”
118            self.info["info"] = "Performance not improving"
119            # 然后把终止训练对应的 value 设为 True
120            self.info["terminate"] = True
121            # 并返回所有信息
122            return self.info
123        # 2) 如果过拟合程度超过了阈值
124        if self. descend counter >= self.n tolerance:
125            # 就告诉调用者 “model 已经过拟合了”
126            self.info["info"] = "Over-fitting"
127            # 然后同样把终止训练对应的 value 设为 True
128            self.info["terminate"] = True
129            # 并返回所有信息
130            return self.info
131        # 3) 调用相应方法, 处理是否需要备份最优参数的问题
132        self._handle_is_best()
133        # 4) 调用相应方法, 处理周期性地备份参数的问题
134        self._handle_period(last_score)
135        # 返回所有信息
136        return self.info

```

可以看到, 我们确实已经将很多部分抽成了单独的方法以便理解, 下面就来一一展示这些部分的实现。首先来看唯一可以算作算法的部分, 即用于维护 **rm** 与 **rs** 的方法的代码。

```

137     def update running info(self, last score, n history):
138         # 如果实际滑动窗口长度还没有达到预设的滑动窗口长度

```

```

139     # 又或者实际滑动窗口长度与已有 score 的总数恰好一致时
140     if n history < self.n history or n history == len(self. scores):
141         # 1) 如果还没有进行初始化的话, 就进行相应的初始化
142         # 注意我们是跳过了第一次的, 所以初始化时
143         # 要把前两次的 score 一起算
144         if self. running sum is None:
145             self. running sum = self. scores[0] + self. scores[1]
146             self. running square sum = (
147                 self. scores[0] ** 2 + self. scores[1] ** 2)
148         # 2) 如果已经初始化了的话, 就直接加上最新的 score 即可
149         else:
150             self. running sum += last score
151             self. running square sum += last score ** 2
152     # 如果实际滑动窗口长度已经达到了预设的长度的话
153     # 就通过递推公式维护滑动求和与滑动平方和
154     else:
155         previous = self. scores[-n history - 1]
156         self. running sum += last score - previous
157         self. running square sum += last score ** 2 - previous ** 2
158     # 如果还没有初始化历史最优表现的话, 就初始化相应的属性
159     # 注意这里的 improvement 是“当前表现相对于历史最优表现的提升”
160     # 而且如果当前表现不是历史最优表现的话, improvement 就会是 0
161     if self. running best is None:
162         if self. scores[0] > self. scores[1]:
163             improvement = 0
164             self. running best, self. is best = self. scores[0], False
165         else:
166             improvement = self. scores[1] - self. scores[0]
167             self. running best, self. is best = self. scores[1], True
168     # 否则, 根据当前表现和历史最优表现的对比来更新相应的属性
169     elif self. running best > last score:
170         improvement = 0
171         self. is best = False
172     else:
173         improvement = last_score - self._running_best
174         self._running_best = last_score
175         self._is_best = True
176     # 返回当前表现相对于历史最优表现的提升 (若没有提升, 则将返回 0)
177     return improvement

```

可以看到, 虽然维护 `rm` 与 `rs` 在一开始看来还挺复杂, 但是当把它们递推公式写出来之后, 就把这个维护工作转换为了维护滑动求和 (`self._running_sum`) 与滑动平方和 (`self._running_square_sum`), 而这两者的维护是比较简单的。

然后再来看看 `TrainMonitor` 中的核心逻辑——判断 `model` 过拟合程度的两个相关方法。

```

178     # 定义处理正处于过拟合的情形的方法
179     def handle overfitting(self, last score, res, std):
180         # 如果 model 此前没有处于过拟合状态 (过拟合程度为 0) 的话
181         if self._descend_counter == 0:

```

```

182         # 就认为此时应该把今后出现的最优的参数备份下来
183         self.info["save best"] = True
184         # 并用相应属性储存此时的 score
185         self. over fit performance = last score
186         # 根据公式累积过拟合程度
187         self. descend counter += min(self.n tolerance / 3, -res / std)
188         # 将标识“是否处于过拟合状态”的属性设置为 1
189         self. over fitting flag = 1
190
191     # 定义处理正从过拟合状态中恢复的情形的方法
192     def handle recovering(self, improvement, last score, res, std):
193         # 如果 model 的性能有了飞跃式的进步的话
194         # 就认为需要把最优参数备份下来
195         if res > 3 * std and self. is best and improvement > std:
196             self.info["save best"] = True
197         # 计算新的过拟合程度
198         new counter = self. descend counter - res / std
199         # 如果之前正处于过拟合, 而新的过拟合程度不大于 0 的话
200         # 就说明此时 model 完全从过拟合的状态中恢复了回来
201         if self. descend counter > 0 >= new counter:
202             # 于是就需要把一些属性复原
203             self. over fit performance = math.inf
204             if last score > self. best checkpoint performance:
205                 self. best checkpoint performance = last score
206                 if last score > self. running best - std:
207                     self.info["save checkpoint"] = True
208                     self.info["info"] = (
209                         "Current snapshot ({} ) seems to be working well, "
210                         "saving checkpoint in case "
211                         "we need to restore".format(len(self. scores))
212                     )
213             self. over fitting flag = 0
214         # 更新过拟合程度
215         self. descend counter = max(new counter, 0)

```

在处理完过拟合相关的事宜之后, 只需再把“备份最优参数”与“周期性地备份参数”这两个逻辑实现出来即可:

```

216     def _handle_is_best(self):
217         # 如果当前表现是最佳表现的话
218         if self. is best:
219             # 就总是不终止训练
220             self.info["terminate"] = False
221             # 而且如果此时需要备份最优参数的话
222             if self.info["save best"]:
223                 # 就调整 self.info 中的相关信息以告诉调用者
224                 # 当前的模型参数应该进行备份
225                 self.info["save checkpoint"] = True
226                 self.info["save best"] = False
227                 self.info["info"] = (

```



```

228         "Current snapshot ({} leads to best result we've ever had, "
229         "saving checkpoint since {}".format(len(self._scores))
230     )
231     # 可以根据 model 是否处于过拟合状态
232     # 来调整告诉调用者的信息
233     if self.over_fitting_flag:
234         self.info["info"] += "we've suffered from over-fitting"
235     else:
236         self.info["info"] += "performance has improved significantly"
237
238     def handle_period(self, last_score):
239         # 如果到了需要周期性备份参数的时间
240         if len(self._scores) % self.snapshot_ratio == 0:
241             # 而且当前模型表现比之前备份参数时的表现都要好的话
242             if last_score > self._best_checkpoint_performance:
243                 # 就调整 self.info 中的相关信息以告诉调用者
244                 # 当前的模型参数应该进行备份
245                 self._best_checkpoint_performance = last_score
246                 self.info["terminate"] = False
247                 self.info["save_checkpoint"] = True
248                 self.info["info"] = (
249                     "Current snapshot ({} leads to best checkpoint "
250                     "we've ever had, saving checkpoint in case "
251                     "we need to restore".format(len(self._scores))
252                 )

```

以上就是 TrainMonitor 的所有实现。需要指出的是，这些实现并非是唯一正确的，它们只是在上两节给出的笔者自己测试后发现在大多数情景下简单有效的思想下的实现。在某个特定的问题上，同样正确甚至更好的实现是肯定存在的，所以在必要的时候，大家可以琢磨一下最适合手头上的任务的实现。

6.6 本章小结

- 数据的准备不仅需要考虑“干净漂亮”的数据，还需要考虑直接从文件中读来的、可能不太好的数据。
- 数据准备大致可以分为以下三个阶段。
 - stage = 1（第 1 阶段）：接收的是新数据，所以关注的重点是提取相应的信息，并进行非数值型离散型特征的数值化与冗余特征的去除。
 - stage = 2（第 2 阶段）：接收的是旧数据，所以关注的重点是后续的缺失值处理与连续型特征的预处理上。
 - stage = 3（第 3 阶段）：可以视为第 1 阶段和第 2 阶段的糅合。换句话说，先后进行第 1 阶段和第 2 阶段与只进行第 3 阶段是等价的。

- 数据的转换包括两大步骤：
 - 非数值型离散型特征的数值化与冗余特征的去除（第 1 阶段、第 3 阶段）。
 - 缺失值处理与连续型特征的预处理（第 2 阶段、第 3 阶段）。
- 训练的监控需要做到如下两点：
 - 若发现 model 出现过拟合现象，则提前终止训练。
 - 若发现 model 存在欠拟合现象，则延长训练步数。
- 为了更合理地监控训练，我们应该使用更具统计意义的标准（running mean、running std）来衡量当前 model 的表现。

第 7 章

工程化机器学习框架

第 6 章我们介绍了如何将数据准备的诸多工作进行抽象与封装，从而使得相应的实现能够直接用在各种 model 之上。虽然这在理论上已经构成了完整的机器学习流程，但是在实际的（工程化）任务中，还需要做一些额外的工作。比如，如果要在实际任务中在诸多 model 中挑选出一个 model 的话，只用每个 model 做一次实验是肯定不行的，必须进行多次的、不重复的、带有随机性的实验，这样才能获得 model 在数据集上的真实性能。又正如 5.5 节最后所说，虽然 AdvancedNN 拥有诸多可喜的性质，但是“没有免费的午餐”定律却注定了某个具体的模型是无法在所有问题上都做到最好的。因此，提供一个参数搜索的功能就会显得至关重要。不过，由于 AdvancedNN 的诸多技术确实非常有效，所以虽然本章将介绍的参数搜索方法会非常朴素平凡，但是它在实际任务的表现中却非常好。

本章主要涉及的知识点有：

- logging 的简要使用
- 多次实验的实现
- 参数搜索的实现
- 真实数据集下的应用

7.1 输出信息的管理

在本章之前的代码实现中，但凡想要输出机器学习模型的某些中间信息（比如模型当前的表现等），我们都是直接采用了 Python 自带的 print 方法来将信息输出到 console 中。然而，当我们想要知道的信息很多时，这种方式就会在 console 中进行大量的输出，导致其中一些特别重要的信息被淹没。

举一个例子，此前我们实现分阶段数据预处理时，时不时会使用 `print` 函数来输出数据预处理的进度，其在多次实验中的实际效果如图 7.1 所示。

```
Fetching data
Fetching data
Fetching data
Generating data info
All values in column 26 are the same, it'll be treated as redundant
All values in column 27 are the same, it'll be treated as redundant
Transforming train data at stage 1
These 2 columns are redundant: [26, 27], they will be removed
Transforming test data at stage 1
Transforming train data at stage 2
Transforming test data at stage 2
Training k-random with k=3, cv_rate=0.1 and test_rate=0.1
Transforming train0 data at stage 2
Transforming cv0 data at stage 2
Sample weights will be used since class_ratio < 0.1 (0.064831)
Sample weights are not provided, they'll be generated automatically
```

图 7.1 多次实验中输出到 console 的信息（节选）

虽然这些信息确实能够让我们更好地理解数据集的特性，能让我们更好地把握程序运行的进度，但是当对同一个实验重复进行很多次时，这些信息就会使 `console` 变得十分拥挤，而且最有价值的信息（比如，多次实验中每次实验结束后模型的表现）就会难以搜寻。为此，对输出信息进行管理在工程化的实现中是非常有必要的。一个良好的实现范式就是将这些有用但却没那么重要的信息都写进一个文件里，在 `console` 中则只保留简洁的、重要的信息。对于这个记录了各种各样程序运行信息的文件，我们通常会称它为“日志文件（log file）”。而在 Python 中管理日志文件的方法，就是使用 `logging` 这个标准库。

`logging` 的使用方法有很多，本节我们会介绍一种简单通用的方式。具体而言，我们会实现一个叫 `LoggingMixin` 的类，这个类能够直接被任何一个已有的类（比如某个机器学习 `model`）继承，从而使得目标类获得方便的、将信息记录在日志文件中的方法。

在介绍 `LoggingMixin` 的具体实现之前，我们需要先知道 `logging` 的使用范式，即需要知道在什么场景下应该使用 `print`，在什么场景下应该使用 `logging`，使用 `logging` 时又应该使用哪种“级别（level）”的 `logging`。`logging` 的使用范式在 Python 官网上有比较详细的介绍（<https://docs.python.org/3/howto/logging.html#logging-basic-tutorial>），这里笔者就择其重点进行翻译，对于想了解更多细节的读者可以直接参见上述链接。

- 使用 `print` 的场景：普通的命令行脚本或命令行程序的输出。
- 使用 `logging` 的 `info` 级别的场景：程序正常运行时的信息输出。
- 使用 `logging` 的 `debug` 级别的场景：程序运行时的细节信息输出。
- 使用 `logging` 的 `warning` 级别的场景：程序在运行中遇到不完全正常却又不一定导致严重错误的情况时的信息输出。
- 使用 `logging` 的 `error` 级别的场景：遇到严重错误，却又不能直接中断程序运行时的信息输出。

注意：`logging` 中的级别可以认为是有序的：`debug < info < warning < error`。此外，`logging` 中还有一个 `critical` 级别，它是比 `error` 还严重的级别，使用频率较低。

结合这些使用范式不难看出，此前我们进行各种代码实现时 `print` 出来的信息基本都应该改为使用 `logging` 的 `debug` 级别或 `info` 级别（如果对应的信息特别重要的话）来输出。为了使得这种改动不至于“伤筋动骨”，我们需要保证日志管理类 `LoggingMixin` 的强拓展性与功能的齐全性，因此我们至少需要实现如下 4 个功能：

- 初始化进行 `logging` 所需的各种环境。
- 根据名字（`name`）和日志文件所在的路径（`file`）来获取一个能进行日志记录的工具（`logger`）。
- 利用 `logger` 来将某条信息写入日志文件（`log_msg`）。
- 利用 `logger` 来将某整块信息写入日志文件（`log_block_msg`）。

这 4 个功能分别对应着 4 个方法，下面我们就来逐一展示它们的具体代码。首先来看看 `LoggingMixin` 本身的初始化，如代码 7.1 所示。

代码 7.1 日志管理类 `LoggingMixin` 的实现：DistBase.py

```
01 class LoggingMixin:
02     # 在类里面直接初始化一个全局 logger
03     logger = logging.getLogger("")
04     # 在类里面直接定义一个记录着已有的日志文件名的集合 (set)
05     initialized_log_file = set()
06
07     # 定义一个获取所有日志文件所在的文件夹的 property
08     @property
09     def logging_folder_name(self):
10         # 在这里笔者将日志文件夹统一放在了“./_Tmp/_Logging/xxx”下
11         # 其中 xxx 是相应 model 的名字，大家完全可以根据需要
12         # 更改这里这个文件夹路径
13         folder = os.path.join(os.getcwd(), "_Tmp", "_Logging", self.name)
14         # 如果该文件夹不存在，就创建该文件夹
15         if not os.path.isdir(folder):
16             os.makedirs(folder)
17         return folder
```

以上就是 `LoggingMixin` 的初始化，可以看到它的初始化有一个和之前所有类的初始化都不一样的地方——那就是它没有“`__init__`”方法。这其实也正是它的后缀名为“`Mixin`”的原因，即它必须被某个 `model` 继承而无法脱离 `model`，且作为一个个体来单独运作。事实上，在上述 `logging_folder_name` 这个 `property` 里面我们用到了 `self.name`，而 `LoggingMixin` 本身是没有 `name` 这个属性的，这个属性必须由 `model` 来提供。再比如我们马上就会用到的 `loggers` 属性，它是一个储存所有 `logger` 的字典的非常重要的属性，不过 `LoggingMixin` 本身仍然没有这个属性，它仍然必须由 `model` 来提供。

注意：在第 6 章实现半自动化机器学习框架时，我们曾经定义过许多数据信息与数据本身的缓存路径以管理数据的准备过程。不难看出，如果应用 `Mixin` 的思想的

话，这些缓存路径都应该抽象成一个 Mixin 中的 property，以方便后续的各种拓展。事实上，我们在后文会直接调用这些 property，所以感兴趣的读者可以先行参阅附录 F 中相应的说明，也可以直接参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/DistBase.py 中一开始的 Data-CacheMixin 部分。不过由于在必要的场合我们都会有详尽的注释，所以即使不先行了解这里面的细节也不会对理解核心思想造成困难。

在完成了 LoggingMixin 本身的初始化之后，我们就要来说明如何初始化进行 logging 所需的环境了。虽然以下即将展示的代码背后有着比较复杂的底层逻辑，不过在笔者看来，只从直观上去理解它们的表层含义即可，对于底层的東西不求甚解可能反而是更高效的选择。

```

18     def init_logging(self):
19         # 这里的 self.loggers 即为上文提到的储存所有 logger 的字典
20         # 如果 loggers 不是 None，说明已经进行过初始化
21         # 从而直接终止即可
22         if self.loggers is not None:
23             return
24         # 否则，定义一个管理输出到 console 的信息的“管理者”
25         console = logging.StreamHandler()
26         # 将该管理者的“级别”设置为 info 级别
27         # 这意味着只有 ≥ info 级别的信息才会被输出到 console 中
28         # （即 debug 级别的信息不会被输出到 console 中）
29         console.setLevel(logging.INFO)
30         # 定义一种信息格式，该格式的具体内容为：
31         # 1）前 20 个字符是进行输出的 logger 的名字（name）
32         # 2）然后是 8 个字符的级别名字（levelname）（比如“INFO”）
33         # 3）最后则是输出信息的主体（message）
34         formatter = logging.Formatter(
35             "%(name)20s - %(levelname)8s - %(message)s"
36         )
37         # 将输出到 console 中的信息的格式设置为上述信息格式
38         console.setFormatter(formatter)
39         # 获取全局 logger
40         root_logger = logging.getLogger("")
41         # 清除全局 logger 的所有已有的“管理者”
42         root_logger.handlers.clear()
43         # 将全局 logger 的级别设置为 debug
44         root_logger.setLevel(logging.DEBUG)
45         # 同时把 console 的管理者加入全局 logger 的管理者行列中
46         # 需要指出的是，全局 logger 本身的级别为 debug
47         # 不会影响 console 管理者的级别为 info 这个事实
48         root_logger.addHandler(console)
49         # 将 loggers 初始化为空字典
50         self.loggers = {}

```

上述代码就完成了进行 logging 的准备，它所做的事概括起来其实就是如下两件：

- 将所有 logger 的默认级别都设置为 debug。
- 将输出到 console 的信息的级别设置为 info。

这样，只要我们把所有有用但又不是特别重要的信息的级别都设置为 debug，然后把所有特别重要的信息的级别都设置为 info 的话，我们就不仅能保证 console 中信息的高质量，还能保证没有信息损失——因为 logger 能将相应的信息输出到日志文件中，所以如果确实有需要的话，只需查阅相应的日志文件即可。

接下来就要实现获取 logger 的方法了。由于 logging 中已经有相应的实现，所以只需做一些简单的封装即可。

```

51     # 定义获取 logger 的方法
52     # 参数 name 代指该 logger 的名字
53     # 参数 file 代指该 logger 对应的日志文件的名字
54     def get_logger(self, name, file):
55         # 如果 loggers 中已经有相应的 logger 的话
56         # 就直接返回相应 logger
57         if name in self.loggers:
58             return self.loggers[name]
59         # 利用 logging folder name 这个 property 来获取日志文件所在的文件夹
60         folder = self.logging_folder.name
61         # 根据上述文件夹和日志文件的名字来获取日志文件的路径
62         log_file = os.path.join(folder, file)
63         # 如果该日志文件没有被初始化过
64         if log_file not in self.initialized_log_files:
65             # 就创建一个相应的空白日志文件
66             with open(log_file, "w"):
67                 pass
68             # 并把该日志文件加入“已被初始化过的日志文件集合”中
69             self.initialized_log_files.add(log_file)
70         # 定义该日志文件对应的“管理者”
71         # 第二个参数是“a”的意思是，每次将信息写入日志文件时
72         # 在日志文件中“接着写（append）”这些信息
73         # 而不是在日志文件中“从头写”这些信息
74         # 如果想要“从头写”的话，把“a”改成“w”即可
75         log_file = logging.FileHandler(log_file, "a")
76         # 将日志文件的级别设置为 debug 级别
77         log_file.setLevel(logging.DEBUG)
78         # 定义一种信息记录格式
79         # 后文会给出相应的截图来说明该格式的具体形式
80         formatter = logging.Formatter(
81             "%(asctime)s - %(name)20s - %(levelname)8s - %(message)s",
82             "%Y-%m-%d %H:%M:%S"
83         )
84         # 将日志文件的信息格式设置为上述格式
85         log_file.setFormatter(formatter)
86         # 利用 logging 的相应方法，获取名字为 name 的 logger
87         logger = logging.getLogger(name)

```

```

88     # 将该 logger 的“管理者”设置为 log file
89     logger.addHandler(log file)
90     # 将该 logger 记录进 loggers 中并返回
91     self.loggers[name] = logger
92     return logger

```

至此我们就有了 logging 所需的环境与进行日志记录的工具 logger。在这两者的基础上，进行日志记录就只是调用 logging 中自带的方法而已了。

```

93     # 定义进行单条信息日志记录的方法，各参数的具体含义如下：
94     # 1) msg: 具体的信息
95     # 2) level: 该信息对应的级别，默认为 debug 级别
96     def log_msg(self, msg, level=logging.DEBUG, logger=None):
97         # 如果 logger 是 None，就用全局 logger (self.logger) 来作为 logger
98         logger = self.logger if logger is None else logger
99         # 如果 logger 是 print，就使用 print 方法输出信息
100        # 否则，就用 logger 自带的 log 方法，按级别 (level) 进行日志记录
101        print(msg) if logger is print else logger.log(level, msg)
102
103    # 定义进行整块信息的日志记录的方法，各参数具体含义如下：
104    # 1) title: 这块信息的标题
105    # 2) header: 这块信息的抬头
106    # 3) body: 这块信息的主要内容
107    # 4) level、logger: 和上面 log_msg 方法中的含义一致
108    def log_block_msg(self, title="Done", header="Result", body="",
109                     level=logging.DEBUG, logger=None):
110        # 利用各个参数，获取真正需要进行日志记录的信息
111        msg = title + "\n" + "\n".join(["=" * 100, header, "-" * 100])
112        if body:
113            msg += "\n{}\n".format(body) + "-" * 100
114        # 利用 log_msg 方法进行日志记录
115        self.log_msg(msg, level, logger)

```

以上我们就完成了 LoggingMixin 的所有实现，下面来看看它的具体应用方法和应用之后的具体效果。注意，在展示代码 7.1 之前曾经说过，本章之前所展示的代码实现中 print 出来的信息基本都应该改为使用 logging 的 debug 级别来输出，所以需要之前的代码做出全面而细致的改动。不过，由于我们实现了 LoggingMixin 这个辅助 logging 的类，所以虽然这项重构工程比较“浩大”，但实际的工程量却不多——大致上只需把 print 替换成 self.log_msg 或 self.log_block_msg 方法即可。

所有的重构代码都可以直接在 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/DistBase.py 这个链接中查询到，这里我们选出具有代表性的 TensorFlow 模型的基本框架(Base)来说明如何利用 LoggingMixin 来重构代码。由于只需对 print 函数做出替换，所以下面我们只展示进行了变动的部分，其余部分则会用省略号代替；同时，为了方便大家进行比较，我们会在注释中附上原代码，如代码 7.2 所示。

代码 7.2 重构后的 TensorFlow 基本框架: DistBase.py

```

01 # 继承 LoggingMixin 以便于进行日志记录
02 class Base(LoggingMixin):
03     signature = "Base"
04
05     def __init__(self, name=None,
06                 model_param_settings=None, model_structure_settings=None):
07         .....
08         # 在初始化方法中定义 loggers 属性
09         self.loggers = None
10         # 同时进行 logging 环境的初始化
11         self._init_logging()
12
13     def _snapshot(self, i_epoch, i_iter, snapshot_cursor):
14         .....
15         msg = (
16             "Epoch {:6}  Iter {:8}  Snapshot {:6} ({} ) - "
17             "Train : {:8.6f}  Test : {}".format(
18                 i_epoch, i_iter, snapshot_cursor,
19                 self._metric_name, train_metric,
20                 "None" if test_metric is None else "{:8.6f}".format(test_metric)
21             )
22         )
23         # 原实现是“print(msg)”，这里则先调用相应方法获得一个
24         # 名字为“_snapshot”、对应的日志文件名为“general.log”的 logger
25         # 然后再利用这个 logger 对相应的信息 (msg) 进行日志记录
26         logger = self.get_logger("_snapshot", "general.log")
27         self.log(msg, logger=logger)
28         return train_metric, test_metric
29
30     def save(self, run_id=0, path=None):
31         .....
32         # 这里的原实现为“print("Saving model")”
33         logger = self.get_logger("save", "general.log")
34         self.log_msg("Saving model", logger=logger)
35         with self._graph.as_default():
36             saver = tf.train.Saver()
37             self.save_collections(folder)
38             saver.save(self.sess, os.path.join(folder, "Model"))
39             # 这里的原实现为“print("Model saved to " + folder)”
40             self.log_msg("Model saved to " + folder, logger=logger)
41         return self

```

以上就是继承了 LoggingMixin 之后，model 应该如何利用相应的方法来进行日志记录的方式。大体上来说，我们只在原实现中出现了 print，但事实上应该使用 logging 来进行信息管理的地方做了如下两步：

- 利用 self.get_logger 方法获取相应名字的 logger。同时，考虑到现在 model 的输出还没

有多到需要仔细分类别管理的程度，所以我们将信息都输出到 `general.log` 中，即我们总会把“`general.log`”作为 `self.get_logger` 方法的第二个参数输入。

- 利用 `self.log_msg` 或 `self.log_block_msg` 进行日志记录。上述两个例子涉及的信息都比较简短，所以我们都用了 `log_msg` 方法；对于比较长的信息，就需要用 `log_block_msg` 方法了。

那么在完成所有的重构之后，相应的日志文件（`general.log`）到底会保存在何处，以及日志文件中到底会记录怎样的信息呢？由于文字的描述始终有些不够直观，我们就直接使用截屏来进行说明，如图 7.2~图 7.5 所示。



图 7.2 `general.log` 所在路径（1）

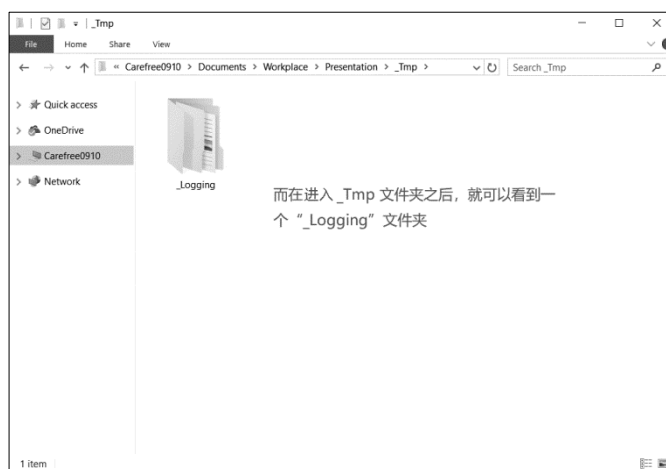


图 7.3 `general.log` 所在路径（2）



图7.4 general.log所在路径（3）

```

2018-01-19 13:38:50 - _get_data_from_file - DEBUG - Fetching data
2018-01-19 13:38:50 - _get_data_from_file - DEBUG - Fetching data
2018-01-19 13:38:50 - _get_data_from_file - DEBUG - Fetching data
2018-01-19 13:38:50 - _load_data - DEBUG - Generating data info
2018-01-19 13:38:50 - _load_data - DEBUG - All values in column 26 are the same, it'll be treated as redundant
2018-01-19 13:38:50 - _load_data - DEBUG - All values in column 27 are the same, it'll be treated as redundant
2018-01-19 13:38:50 - _transform_data - DEBUG - Transforming train data at stage 1
2018-01-19 13:38:50 - _transform_data - DEBUG - These 2 columns are redundant: [26, 27], they will be removed
2018-01-19 13:38:50 - _transform_data - DEBUG - Transforming test data at stage 1
2018-01-19 13:38:50 - _transform_data - DEBUG - Transforming train data at stage 2
2018-01-19 13:38:50 - _transform_data - DEBUG - Transforming test data at stage 2
2018-01-19 13:38:50 - _k_series_process - DEBUG - Training k-random with k=3, cv_rate=0.1 and test_rate=0.1
2018-01-19 13:38:50 - _transform_data - DEBUG - Transforming train0 data at stage 2
2018-01-19 13:38:50 - _transform_data - DEBUG - Transforming cv0 data at stage 2
2018-01-19 13:38:50 - _handle_unbalance - DEBUG - Sample weights will be used since class_ratio < 0.1 (0.064831)
2018-01-19 13:38:50 - _handle_unbalance - DEBUG - Sample weights are not provided, they'll be generated automatically

```

图7.5 （多次实验中）general.log记录的信息（节选）

可以看到，假设某个代码文件（比如图 7.2 中的 Test.py）调用了某个继承 LoggingMixin 的 model 的话，那么 general.log 就会位于该代码文件路径下的“_Tmp”文件夹中的“_Logging”文件夹中的相应文件夹中，这与代码 7.1 中的相应实现是一致的。同时也可以看到，图 7.5 所示的 general.log 中记录的信息和图 7.1 所示的输出到 console 中的信息没有任何不同（除了 general.log 中还记录了当前信息生成时所处的时间、相应的 logger 的名字以及信息所对应的级别以外），这说明 general.log 确实能够满足我们的需求。

以上就是 LoggingMixin 的具体实现、应用方式与应用效果。最后需要指出的是，虽然我们一直说 LoggingMixin 要配合 model 来使用，但是从代码实现层面上不难看出，只要某个类拥有 name 和 loggers 这两个属性，那么该类就可以通过继承 LoggingMixin 来获得进行日志记录的诸多功能。

7.2 多次实验的管理

7.1 节实现了能够帮助我们管理输出信息的 LoggingMixin，从这一节开始我们就要正式着手实现工程化机器学习框架（DistMixin）本身了。正如本章开头所说，在实际的（工程化）任务中，挑选 model 的标准必须是 model 在多次不重复的、带有随机性的实验（后简称“多次实验”）

上的表现，而不能简单地根据单次实验中 model 的表现来进行筛选，所以实现多次实验的管理是 DistMixin 的当务之急。

注意：虽然 7.1 节实现的 LoggingMixin 的应用范围很广（只要求目标类拥有 name 属性与 loggers 属性），但是从本节开始实现的 DistMixin 的应用范围则局限在基于第 6 章实现的半自动化机器学习框架（AutoBase）拓展的模型上。这主要是因为工程化的实现一般都会要求做到数据准备方面的自动化，所以我们才做了这样的约束。但事实上，即使有这样的约束，从附录 A 中的相应实现中也可以看出，只要某个模型算法（比如 LinearSVM）能够写成基于梯度下降法的形式，那么它就能在不改动一行代码的情况下简单地应用上 DistMixin。

不过，虽然进行多次实验从直观上来说非常简单，似乎我们只需要在训练步骤的外面套一个循环的逻辑（比如 for 循环）就可以了，然而如果考虑到实际任务的多样性以及整个框架的易用性、可拓展性的话，多次实验的管理就成为比想象中要困难不少的任务。

虽然多次实验是一个很直观的概念，不过考虑到严谨性，在展开说明多次实验的管理思路与具体代码实现之前，我们还是先用规范一点的语言来叙述一下多次实验的算法框架，如算法 7.1 所示。

算法 7.1 多次实验

输入：某个待评估的模型 G ，评估函数 s ，数据集 X ，重复实验的次数 k

过程：

(1) 根据某种划分方法，将数据集 X 划分为训练用数据集 X_{tr} 与测试用数据集 X_{te} ：

$$X \rightarrow X_{tr} \cup X_{te}, (X_{tr} \cap X_{te} = \emptyset)$$

(2) 令 score_sum 为 0，并将如下步骤重复 k 遍：

a) 初始化 G 的所有参数，保证上一次实验不会影响这一次的实验

b) 根据某种划分方法，将训练用数据集划分为训练集和交叉验证集

$$X_{tr} \rightarrow X_{train} \cup X_{cv}, (X_{train} \cap X_{cv} = \emptyset)$$

c) 利用训练集和交叉验证集，训练模型 G

d) 利用评估函数 s ，算出训练好的模型 G 在整个数据集 X 上的分数并累加到 score_sum 中

$$\text{score_sum} \leftarrow \text{score_sum} + s(G; X)$$

输出：模型 G 在数据集 X 上的综合表现 G_{score} ，其中

$$G_{\text{score}} \triangleq \frac{1}{k} \cdot \text{score_sum}$$

由上述算法不难看出，如果想要完善地实现多次实验的管理的话，至少需要达成如下两点要求：

- 保证 G 的初始化方法足够彻底，从而使得多次实验中的每两次实验之间都是相互独立、互不影响的。
- 保证多次实验在代码实现上的差异仅仅体现在算法 7.1 中的 (1) 和 (2) 的 b) 处；这是因为从算法 7.1 中不难看出，不同的多次实验方法之间的差异仅仅体现在不同的数据集划分方法上。

同时如果考虑到工程上的需求的话，我们还需要做到如下两点：

- 保证多次实验的总耗时以及每次实验的耗时都是可控的。否则如果一项实际任务有时时间限制的话，这整套框架可能就会完全派不上用场。
- 不仅将最终结果进行输出，而且也要输出多次实验中每次实验的结果，从而能更直观地感受模型的稳定性；同时，不只是单纯地把实验结果输出到 console 中，而是要将结果写入文件以方便查看及进行后续的处理。

在上面的 4 个要求中，解决前 3 个要求是本节所主要关心的问题，至于第 4 个要求则恰恰被上一节解决了大半，所以本节只需在上一节相应的方法外做一些封装即可。

首先是第 1 个要求。由于我们规定了 DistMixin 必须在 AutoBase 的基础上进行应用，所以模型 G 其实必须是基于 TensorFlow 写的模型，这就使 G 的参数的初始化变得非常平凡——只需调用之前也用到过的 TensorFlow 自带的函数即可。

```
01 def reset_all_variables(self):
02     # 将默认的计算图设置为当前所用的计算图 self._graph
03     with self._graph.as_default():
04         # 利用 tf.global_variables_initializer() 来
05         # 初始化所有可训练变量 variable
06         self._sess.run(tf.global_variables_initializer())
```

注意：上述代码以及接下来与工程化机器学习框架相关的代码都是 DistMixin 的实现中的一部分。不过由于 DistMixin 与 LoggingMixin 类似，都需要在某个类的基础上发挥作用，所以 DistMixin 本身是没有初始化方法的。也正因此，我们只会根据具体场景来介绍它的各个方法。如果对完整的实现感兴趣的话，则可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/DistBase.py 中的 DistMixin 部分，笔者个人也建议大家将完整的代码下载下来并与本节的内容进行对照。

然后来看看第 2 个要求，该要求的实现也正是我们管理多次实验的核心。不难想象，为了达成第 2 个要求，我们必须把算法 7.1 中的各个步骤都抽象成独立的方法，从而得以只通过更改其中划分数据对应的方法来更改多次实验的方法。

能够做到这点的抽象方式有很多，这里我们就只叙述其中一种非常直观但也非常有效的方式。具体而言，我们会将多次实验的过程（`k_series_process`）分离为如下 4 个步骤。

- 多次实验的初始化（`k_series_initialization`）：对应着算法 7.1 的步骤（1），它能够把数据集 X 划分为 X_{tr} 和 X_{te} ，同时它还会针对其他一些需求来做准备。
- 多次实验中的数据划分（`cv_method`）：对应着算法 7.1 的步骤（2）中的 b），它能够把训练用数据集 X_{tr} 划分为 X_{train} 和 X_{test} 。
- 多次实验中模型 G 表现的评估（`k_series_evaluation`）：对应着算法 7.1 的步骤（2）中的 d），用于评估 G 在单次实验中的表现以及在整个多次实验过程中的表现。
- 多次实验的收尾（`k_series_completion`）：对应着算法 7.1 的输出步骤，能够将所需的信息进行记录，并将模型 G 的状态还原到进行多次实验之前的状态。

我们接下来会先在第 1 节实现多次实验的整体框架（`k_series_process`），然后会在后续的 4 节中分别对上述 4 个步骤的原理与具体代码实现进行说明。

7.2.1 多次实验的框架

为了让大家对多次实验的实现有一个“大局观”，在本节中，我们会先展示多次实验整体框架的实现。虽然其中会用到许多后续章节里面才会介绍的方法，但是由于相应方法的功能已经在算法 7.1 中进行了说明，而且整体框架的实现本身其实就是对算法 7.1 的“代码翻译”，所以不会对理解造成太大困难。

不过虽说如此，整体框架总代码量还是比较大的，为此我们会将代码分成两部分进行说明。首先来看第一部分，也是进入多次实验主循环之前的实现。

```

01     # 能够获取多次实验当前总耗时的 property
02     # 其中，self._k_series_t 是记录着多次实验开始时的时间的属性
03     @property
04     def k series time delta(self):
05         return time.time() - self.k series t
06
07     """
08     多次实验的整体框架
09     k: 进行多次实验的次数
10     data: 数据集 X
11     cv_rate: 将 $X_{tr}$ 划分为 $X_{train}$ 和 $X_{cv}$ 时， $X_{cv}$ 样本数所占的比例
12     test_rate: 将 X 划分为 $X_{tr}$ 和 $X_{te}$ 时， $X_{te}$ 样本数所占的比例
13     sample_weights: 样本权重
14     msg: 进行多次实验之前想要输出的信息
15     cv_method: 将 $X_{tr}$ 划分为 $X_{train}$ 和 $X_{cv}$ 的方法
16     kwargs: 储存多次实验中模型 G 的超参数的字典
17     """
18     def _k_series_process(self, k, data, cv_rate, test_rate, sample_weights,
19                          msg, cv method, kwargs):
20         # 利用多次实验的初始化方法 k_series_initialization，划分出 $X_{tr}$ 和 $X_{te}$ 
21         # 同时，也获取 names（多次实验中各个数据集的名字）这个信息
22         # 该方法的详细说明会在下一节中给出
23         info = self.k series initialization(
24             k, data, test_rate, kwargs.pop("time_limit", -1))
25         x 1, y 1, x test 2, y test 2, names = info
26         # 从 kwargs 中提取时间限制的设置，单位为秒，默认为-1（即没有限制）
27         time limit = kwargs.pop("time limit", -1)
28         # 获取一个名字为“_k_series_process”的 logger
29         # 它对应的日志文件仍为 general.log
30         logger = self.get_logger("_k_series_process", "general.log")
31         # 检查上述初始化方法的耗时有没有超过时间限制
32         if 0 < time limit <= self.k series time delta:
33             # 如果已经超过了的话，就告诉调用者
34             # “多次实验尚未开始，耗时就已经超过时间限制了”
35             self.log_msg(

```

```

36         "Time limit exceeded before k_series started", logger=logger)
37         # 然后直接终止
38         return
39     # 调用相应的方法, 处理参数搜索相关的事宜
40     # 该方法的具体说明会在后文介绍参数搜索时进行
41     time limit = self. handle param search time limit(time limit)
42     # 根据 cv_rate 和  $X_{tr}$ , 获取  $X_{cv}$  的样本数
43     n cv = int(cv rate * len(x 1))
44     print_settings = True
45     # 如果提供样本权重的话
46     # 就把 self._sample_weights 设置为相应的权重
47     if sample_weights is not None:
48         self._sample_weights = np.asarray(sample_weights, np.float32)
49     # 用一个变量来储存当前的样本权重
50     sample_weights_store = self._sample_weights
51     # 利用 self.log_msg 方法, 输出 msg 信息
52     self.log_msg(msg, logger=logger)
53     # 用变量 all_idx 来储存  $X_{tr}$  中所有样本打乱后的“下标”
54     all_idx = np.random.permutation(len(x_1))

```

以上就是 `k_series_process` 的前半部分, 其中第 25 行处的变量名可能会让大家感到有疑惑: 为什么我们要取 `x_1`、`y_1` (对应着 X_{tr}) 和 `x_test_2`、`y_test_2` (对应着 X_{te}) 这种奇怪的名字呢? 这是由于我们的数据有可能是直接从文件中读取而来的, 如果多次实验中的每次实验都从头读取文件并从头预处理数据的话, 会造成大量的浪费。不过, 得益于上一章对数据预处理的过程分了阶段, 所以我们就可以通过如下流程来减少浪费:

- 将数据集 X 划分为 X_{tr} 和 X_{te} 时, 仅进行第 1 阶段的预处理。
- 对 X_{te} 以及接下来每次单独实验中划分出来的 X_{train} 和 X_{cv} 进行第 2 阶段的预处理。

这样, 从文件中读取数据的步骤 (第 1 阶段的数据预处理) 就只用进行一次, 这样就最大程度地减少了浪费。在这种处理下, 上述奇怪的命名的意义就明确了:

- `x_1`、`y_1` 对应着 X_{tr} 在经过了第 1 阶段的预处理后得到的特征向量与标签。
- `x_test_2`、`y_test_2` 对应着 X_{te} 在经过了第 1 阶段和第 2 阶段的预处理后得到的特征向量与标签。

在做好前半部分的准备后, 我们来看看后半部分主循环的实现。

```

55     for i in range(k):
56         # 如果 self. sess 不是 None 的话, 就初始化所有参数
57         if self. sess is not None:
58             self.reset all variables()
59         # 用 skip 变量来记录 “是否需要跳过当前实验”
60         skip = False
61         while True:
62             # 利用划分  $X_{train}$  和  $X_{cv}$  的方法获取当前实验的数据信息
63             # 具体划分方法的说明会在第 3 节中给出
64             rs = cv method(x 1, y 1, n cv, i, k, all_idx)
65             # 如果划分是成功的, 就把相应的数据记录在各个变量中

```

```

66         if rs["success"]:
67             x_train, y_train, x_cv, y_cv, train_idx = rs["info"]
68             break
69         # 如果不成功而且需要“重试”的话，就重来一遍
70         if rs["info"] == "retry":
71             continue
72         # 如果不成功而且不需要重试的话，就说明当前实验需要被跳过
73         # 而之所以会出现需要跳过当前实验的情况的原因
74         # 我们同样会在第 3 节中给出
75         x_train = y_train = x_cv = y_cv = train_idx = None
76         skip = True
77         break
78     if skip:
79         continue
80     # 如果当前实验没有被跳过的话，就根据 $X_{\text{train}}$ 中各个样本的下标
81     # 提取出 $X_{\text{train}}$ 的样本权重
82     if sample_weights is not None:
83         self.sample_weights = sample_weights_store[train_idx]
84     else:
85         self.sample_weights = None
86     # 进行相应的参数设置
87     kwargs["print_settings"] = print_settings
88     kwargs["names"] = names[i]
89     # 注意要把数据预处理的阶段设置为 2
90     # 因为之前已经经过了第 1 阶段的数据预处理
91     self.data_info["stage"] = 2
92     # 调用 fit 方法进行单次实验
93     self.fit(
94         x_train, y_train, x_cv, y_cv,
95         timeit=False, time_limit=time_limit, **kwargs)
96     # 调用 k series evaluation 方法来完成模型的评估
97     # 该方法的详细说明会在第 4 节中进行，这里只需知道：
98     # 如果该方法返回了 True，就意味着耗时已经超过了时间限制
99     if self._k_series_evaluation(i, x_test_2, y_test_2, time_limit):
100         # 所以就需要 break 出多次实验的循环
101         break
102     print_settings = False
103     # 调用 k_series_completion 方法，完成多次实验的收尾工作
104     # 该方法的详细说明会在第 5 节中给出
105     self._k_series_completion(
106         x_test_2, y_test_2, names, sample_weights_store)
107     return self

```

以上就是多次实验总框架的实现。抛开具体的细节不谈，整体上来说基本和算法 7.1 有着很强的对应关系。而在接下来的 4 节中，我们会逐一补齐其中各个没有展开叙述的部分。

7.2.2 多次实验的初始化

本节会说明如何实现多次实验中的初始化，它对应着算法 7.1 中的第（1）步。概括来说，

我们需要做到这样两点：

- 初始化记录多次实验结果所需的属性与模型 G 的各个参数。
- 从文件中读取数据，进行相应的数据预处理并予以缓存。

具体的代码实现如下所示。

```

108     def k series initialization(self, k, data, test rate):
109         # 用 self._k_series_t 记录下多次实验开始的时间
110         self.k series t = time.time()
111         # 将 data_info 中的 test_rate 一项设置为默认的 test_rate
112         self.data info.setdefault("test rate", test rate)
113         # 调用 self.init_data_info 方法，初始化数据超参数
114         self.init data info()
115         # 将 self._k_performances 属性初始化为空 list
116         # 该 list 将会记录多次实验里每次实验中模型的表现
117         self.k performances = []
118         # 初始化两个属性，这两个属性分别记录着
119         # 多次实验中模型  $G$  表现的均值和标准差
120         self.k performances mean = self.k performances std = None
121         # 定义一个参数字典
122         kwargs = {
123             "numerical idx": self.numerical idx,
124             "shuffle": self.data info["shuffle"],
125             "file type": self.data info["file type"]
126         }
127         # 将上述字典传入 self._load_data 方法中
128         # 以从文件中读取（经过第 1 阶段的数据预处理之后的）数据
129         x 1, y 1, x test 1, y test 1 = self. load data(
130             data, test rate=self.data info["test rate"], stage=1, **kwargs)
131         # 如果没有在搜索参数的话，就把相应的数据进行缓存
132         if not self. searching params:
133             train 1 = np.hstack([x 1, y 1.reshape([-1, 1])])
134             test 1 = np.hstack([x test 1, y test 1.reshape([-1, 1])])
135             np.save(self.train data file, train 1)
136             np.save(self.test data file, test 1)
137         # 利用经过了第 1 阶段处理的数据来进行第 2 阶段的处理
138         # 这一步的主要目的是让模型对整个  $X_{tr}$  有一个总体的认知
139         self. load data(
140             np.hstack([x 1, y 1.reshape([-1, 1])]),
141             names=("train", None), test rate=0, stage=2, **kwargs
142         )
143         # 如果没有提供测试数据集的话，就把 x test 2 和 y test 2 置为 None
144         if x test 1 is None or y test 1 is None:
145             x test 2 = y test 2 = None
146         # 否则，调用 self. load_data 方法
147         # 进行第 2 阶段的数据预处理
148         else:
149             x test 2, y test 2, * = self. load data(
150                 np.hstack([x_test_1, y_test_1.reshape([-1, 1])]),

```

```

151         names=("test", None), test_rate=0, stage=2, **kwargs
152     )
153     # 获取多次实验中，每次实验所用的 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 这两个数据集的名字
154     names = [("train{}".format(i), "cv{}".format(i)) for i in range(k)]
155     # 返回所有信息
156     return x_1, y_1, x_test_2, y_test_2, names

```

总的来说，如果对第 6 章叙述的 `self_load_data` 这个 `AutoBase` 中实现的数据预处理方法很熟悉的话，上述实现就是比较直观的。虽然前文的很多地方都已经提到过，不过由于比较重要，所以这里就再次重申一遍： x_1 、 y_1 这两个训练用数据都只经过了第 1 阶段的数据预处理，这是因为多次实验中的每个单次实验会把它们再次分成训练集和交叉验证集，所以每个单次实验中都需要对划分出来的（新的） X_{train} 和 X_{cv} 进行第 2 阶段的数据预处理。但是， $x_{\text{test_2}}$ 、 $y_{\text{test_2}}$ 在每个单次实验中都是不变的，所以可以在一开始就对它们进行第 2 阶段的数据预处理。

7.2.3 多次实验中的数据划分

本节我们来看看如何在多次实验中进行数据划分，即如何将初始化步骤中得到的训练用数据集 X_{tr} 进一步划分为 X_{train} 和 X_{cv} 。如果大家对第 1 章的内容还有印象的话，就会发现这里数据划分的方法在当时的 1.1.2 节的最后介绍交叉验证时就已经给出过了。具体而言，我们一般会使用如下三种划分数据的方式。

- **K 折交叉验证：**将数据集分成 K 份，每次取其中一份作为 X_{cv} ，取其余的作为 X_{train} ，且 K 次实验取的 X_{cv} 各不相同。
- **留一交叉验证：** $K = N$ 时的 K 折交叉验证，其中 N 为 X_{tr} 的总样本量。
- **简易交叉验证：**每次从 X_{tr} 中随机划出 90%的样本作为 X_{train} ，用其余的样本作为 X_{cv} ，重复 K 次。

从实现的角度来看，由于留一交叉验证只是 K 折交叉验证的特例，所以我们只用实现 K 折交叉验证和简易交叉验证即可。首先来看看 K 折交叉验证的代码：

```

157     def _k_fold_method(self, x_1, y_1, *args):
158         # 1) i 代指当前是多次实验中的第 i 次实验（从 0 开始计数）
159         # 2) k 代指  $K$  折交叉验证中的  $K$ 
160         # 3) all_idx 为已经打乱了的所有样本的下标
161         _, i, k, all_idx = args
162         # 定义一个储存结果的字典，将 success 对应的 value 初始化为 True
163         rs = {"success": True}
164         # 获取  $K$  折交叉验证的  $K$  份数据中，每一份数据的样本量
165         n_batch = int(len(x_1) / k)
166         # 根据当前是第 i 次实验来获取 $X_{\text{cv}}$ 中各个样本的下标
167         cv_idx = all_idx[np.arange(i * n_batch, (i + 1) * n_batch)]
168         # 根据当前是第 i 次实验来获取 $X_{\text{train}}$ 中各个样本的下标
169         train_idx = all_idx[[
170             j for j in range(len(all_idx))
171             if j < i * n_batch or j >= (i + 1) * n_batch
172         ]]

```

```

173     # 根据下标来提取出 $X_{cv}$ 和 $X_{train}$ 
174     x_cv, y_cv = x_l[cv_idx], y_l[cv_idx]
175     x_train, y_train = x_l[train_idx], y_l[train_idx]
176     # 调用相应方法来检查当前划分出来的 $X_{train}$ 和 $X_{cv}$ 是否合法
177     # 注意我们传的第2个参数是“skip”，这意味着如果不合法的话
178     # 我们就会直接“跳过”这第i次实验
179     # 该方法的说明马上在后文给出，这里先按下不表
180     self.cv_sanity_check(
181         rs, "skip", train_idx, x_train, y_train, x_cv, y_cv)
182     # 将结果返回
183     return rs

```

以上即为 K 折交叉验证的实现,如果对 K 折交叉验证熟悉的话,上述代码还是比较直观的。下面来看看简易交叉验证的实现,顾名思义,相应的代码也会更简易一些。

```

184     def k_random_method(self, x_l, y_l, *args):
185         # 这里的n_cv代指 $X_{cv}$ 的样本数
186         n_cv, *_ = args
187         # 与 $K$ 折交叉验证一样,定义一个储存结果的字典
188         rs = {"success": True}
189         # 由于简易交叉验证在每次划分 $X_{train}$ 和 $X_{cv}$ 时都是随机划分的
190         # 所以all_idx也要在每个单次实验中重新生成
191         all_idx = np.random.permutation(len(x_l))
192         # 直接根据 $X_{cv}$ 的样本数,获取 $X_{train}$ 和 $X_{cv}$ 中样本的下标
193         cv_idx, train_idx = all_idx[:n_cv], all_idx[n_cv:]
194         # 利用 $X_{tr}$ 获取 $X_{train}$ 和 $X_{cv}$ 
195         x_cv, y_cv = x_l[cv_idx], y_l[cv_idx]
196         x_train, y_train = x_l[train_idx], y_l[train_idx]
197         # 调用相应方法来检查当前划分出来的 $X_{train}$ 和 $X_{cv}$ 是否合法
198         # 注意我们传入的第2个参数是“retry”，这意味着如果不合法的话
199         # 我们就会尝试“重来”这第i次实验
200         self._cv_sanity_check(
201             rs, "retry", train_idx, x_train, y_train, x_cv, y_cv)
202         # 将结果返回
203         return rs

```

以上就是简易交叉验证的实现。在本节的最后,我们来补充说明一下如何检验划分出来的 X_{train} 和 X_{cv} 是否合法,即self_cv_sanity_check方法的具体实现。不过,虽然接下来展现的代码可能看上去有点复杂,但是合法与否的判定标准只有一个:那就是看 X_{train} 和 X_{cv} 的标签种类是否一致。虽然这个判断可以直接利用Python中集合(set)相关的操作来完成,但由于我们的 X_{train} 和 X_{cv} 都是用numpy数组来表示的,所以用numpy中的方法会更快一些。但是作为代价,相应代码的可读性可能也会差一些。

```

204     def cv_sanity_check(self, rs, handler, train_idx,
205         x_train, y_train, x_cv, y_cv):
206         # 如果是回归问题的话,就没有所谓的标签种类的概念
207         # 即划分出来的 $X_{train}$ 和 $X_{cv}$ 总是合法的
208         if self.n_class == 1:

```

```

209     rs["info"] = (x_train, y_train, x_cv, y_cv, train_idx)
210     # 如果是分类问题的话, 就要看 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 的标签种类是否一致
211     else:
212         # 利用 np.unique 方法, 获取 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 的不同标签
213         y_train_unique = np.unique(y_train)
214         y_cv_unique = np.unique(y_cv)
215         # 如果 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 不同标签的个数一致
216         # 且这些标签彼此之间也能对应得上
217         # 就说明 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 的标签种类一致
218         if len(y_train_unique) == len(y_cv_unique) and np.allclose(
219             y_train_unique, y_cv_unique):
220             rs["info"] = (x_train, y_train, x_cv, y_cv, train_idx)
221             # 否则, 就说明 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 的标签种类不一致
222             # 此时划分出来的 $X_{\text{train}}$ 和 $X_{\text{cv}}$ 就是不合法的
223         else:
224             rs["success"] = False
225             rs["info"] = handler

```

以上就是多次实验中数据划分的全部实现, 虽然背后没有什么深奥的原理, 但是想要理清它和多次实验整体框架的联系还是不太容易且非常重要的。

7.2.4 多次实验中的模型评估

在完成了初始化和数据划分的工作之后, 由于我们不用对单次实验本身做出修改 (直接用之前实现的 `fit` 方法即可), 而且单次实验的评估方法也无须变动, 所以只需想清楚如何将多次实验中各个单次实验的评估整合成对多次实验整体的评估即可。而其中的一个直观易懂, 也是我们即将采用的方法就是, 对多次实验中 k 个单次实验的评估结果取均值与标准差, 来作为衡量 `model` 在多次实验中的整体表现的指标。

```

226     # 参数 i 代指当前是第 i 次实验
227     def _k_series_evaluation(self, i, x_test, y_test, time_limit):
228         # 如果 i 是-1的话, 就说明当前是最后一次评估
229         # 换句话说, 此时应该对之前的所有评估做一个汇总
230         if i == -1:
231             # 如果没有提供测试集的话
232             if x_test is None or y_test is None:
233                 # 就只取每次评估中三个 model 表现中的前两个
234                 # 因为前两个 model 表现分别对应着
235                 # 训练集 $X_{\text{train}}$ 和交叉验证集 $X_{\text{cv}}$ 上 model 的表现
236                 # 而最后一个则对应着测试集 $X_{\text{te}}$ 上 model 的表现
237                 valid_performances = [
238                     performance[:2] for performance in self._k_performances]
239             # 否则, 就把每次评估中的所有 model 表现都取出来
240             else:
241                 valid_performances = self._k_performances
242             # 利用 numpy 相关函数, 算出所有表现的均值和标准差
243             performances_mean = np.mean(valid_performances, axis=0)

```

```

244     performances_std = np.std(valid_performances, axis=0)
245     # 利用 self._print_metrics 方法, 获取相应的信息
246     msg = " - Mean | {} \n".format(
247         self._print_metrics(self._metric_name,
248             *performances_mean, only_return=True))
249     msg += " - Std | {} \n".format(
250         self._print_metrics(self._metric_name,
251             *performances_std, only_return=True))
252     # 根据当前是否正在搜索参数
253     # 获取信息的级别与管理信息的 logger
254     if self._searching_params:
255         level = logging.DEBUG
256         logger = self.param_search_logger
257     else:
258         level = logging.INFO
259         logger = self.k_series_logger
260     # 利用 self.log_block_msg 方法
261     # 将与评估相关的信息作为一个整体进行输出
262     self.log_block_msg(
263         "Generating performance summary", body=msg,
264         level=level, logger=logger
265     )
266     # 返回多次评估的均值与标准差
267     return performances_mean, performances_std
268 # 否则, 说明多次实验还未结束
269 # 此时就需根据训练集 $X_{\text{train}}$ 和交叉验证集 $X_{\text{cv}}$ 
270 # 来进行当前(第 i 次)实验的模型评估
271 # 先利用 self._train_generator 和 self._test_generator
272 # 来获取当前(第 i 次)实验的 $X_{\text{train}}$ 与 $X_{\text{cv}}$ 
273 train_data = self._train_generator.get_all_data(return_weights=False)
274 cv_data = self._test_generator.get_all_data(return_weights=False)
275 # 根据数据集提取出相应的特征向量和标签
276 x, y = train_data[..., :-1], train_data[..., -1]
277 x_cv, y_cv = cv_data[..., :-1], cv_data[..., -1]
278 # 定义 msg 来记录将要输出的信息
279 msg = "Performance of run { :2} | ".format(i + 1)
280 print(" - " + msg, end="")
281 # 利用 self._evaluate 方法来获取当前模型的评估
282 self._k_performances.append(
283     self._evaluate(x, y, x_cv, y_cv, x_test, y_test))
284 # 利用该评估来更新输出的信息
285 msg += self._print_metrics(self._metric_name,
286     *self._k_performances[-1], only_return=True)
287 # 利用 self.log_msg 方法输出信息
288 self.log_msg(
289     msg, logging.DEBUG,
290     self.param_search_logger if self._searching_params
291     else self.k_series_logger

```

```

292         )
293         # 返回当前多次实验的耗时是否已经超过时间限制（如果有的话）
294         return self.k_series_time_delta >= time_limit > 0

```

上述代码里面用到了两种 logger——self.k_series_logger 和 self.param_search_logger，它们的实现如下所示。

```

295     @property
296     def k_series_logger(self):
297         # 将 logger 的后缀名称设置为“k_series”
298         name = "{} k series".format(self.name)
299         if name not in self.loggers:
300             self.get_logger(name, "{}.log".format(name))
301         return self.loggers[name]
302
303     @property
304     def param_search_logger(self):
305         # 将 logger 的后缀名称设置为“param_search”
306         name = "{} param search".format(self.name)
307         if name not in self.loggers:
308             self.get_logger(name, "{}.log".format(name))
309         return self.loggers[name]

```

在这种实现下，多次实验里每个单次实验完成之后，都会输出类似于图 7.6 所示的信息。

```

Performance of run 1 | auc - Train : 0.99894337   CV : 0.99550017   Test : 0.98876404
Performance of run 2 | auc - Train : 0.99798344   CV : 0.99242076   Test : 0.98927477
Performance of run 3 | auc - Train : 0.99968827   CV : 0.99026591   Test : 0.99144535
Performance of run 4 | auc - Train : 0.99820666   CV : 0.98632745   Test : 0.98774259
Performance of run 5 | auc - Train : 0.99643307   CV : 0.99297390   Test : 0.98863636
Performance of run 6 | auc - Train : 0.99853417   CV : 0.99623824   Test : 0.98838100
Performance of run 7 | auc - Train : 0.99921610   CV : 0.99129257   Test : 0.99195608
Performance of run 8 | auc - Train : 0.99944156   CV : 0.99869421   Test : 0.99131767
Performance of run 9 | auc - Train : 0.99816126   CV : 0.99811912   Test : 0.99067926

```

图7.6 单次实验的评估信息

在多次实验全部完成之后，还会额外输出如图 7.7 所示的汇总信息。

```

=====
Result
-----
- Mean | auc - Train : 0.99851199   CV : 0.99353693   Test : 0.98979968
- Std  | auc - Train : 0.00092782   CV : 0.00377155   Test : 0.00146750
-----

```

图7.7 多次实验的评估汇总

而且由上面代码 296~301 行处 self.k_series_logger 的实现不难看出，图 7.6 和图 7.7 所示的信息都会被保存在一个叫“xxx_k_series.log”的文件中，其中 xxx 是数据集的名字，且该文件和 general.log 的路径一致。这样一来，如果我们只关心多次实验的结果，就只需关心 xxx_k_series.log 这个文件；而如果还想进一步了解多次实验中的各部分细节的话，就再查看 general.log 文件即可。

注意:self.param_search_logger 是用于记录参数搜索信息的 logger，它会在下一节用到。

7.2.5 多次实验的收尾工作

前4节其实已经完成了多次实验本身的所有实现，不过为了让 `model` 在进行完多次实验之后还能做进一步的工作，我们必须将 `model` 的状态恢复到多次实验之前的状态。具体而言，我们需要做如下两件事。

- 将多次实验的实验结果记录在某些属性中以方便后续使用。
- 将某些因多次实验而临时改变的超参数复原。

具体的代码实现则如下所示。

```

310     def k_series_completion(self,
311                             x_test, y_test, names, sample_weights_store):
312         # 利用 self._k_series_evaluation 方法获取多次实验中
313         # 各个单次实验里面模型表现的均值和标准差
314         performance_info = self._k_series_evaluation(-1, x_test, y_test, None)
315         self.k_performances_mean = performance_info[0]
316         self.k_performances_std = performance_info[1]
317         # 注意，为了在多次实验中减少浪费，我们将数据预处理分成了
318         # 一次第1阶段与多次第2阶段的过程
319         # 但是在单次实验中，我们直接进行一次第3阶段的数据预处理即可
320         # 所以这里要将数据预处理的阶段设置为3
321         self.data_info["stage"] = 3
322         # 由于在多次实验中划分了很多次数据集，且每次划分完数据之后
323         # 都会生成相应的数据预处理器，这些数据预处理器在今后是用不到的
324         # 所以我们要调用上一章介绍过的相应方法来清除它们
325         for name in names:
326             self._pop_preprocessor(name)
327         # 类似的，由于每次划分完数据之后都会更新临时的样本权重
328         # 所以我们要把样本权重恢复成初始值
329         self.sample_weights = sample_weights_store

```

以上就是多次实验的所有实现，虽然它们背后的逻辑很清晰且本身难度较小，但是想要结合 `model` 来进行完善的管理还是需要花不少力气的。

7.3 参数搜索的管理

7.2节的实现使我们已经能够对某个 `model` 进行多次实验并获取综合评估，这意味着参数搜索变成了可行的方案。正如第5章最后一节所介绍的“没有免费的午餐定理”，在实际应用中是没有普适的、最好的神经网络结构的，所以如果真想在任务中获取最佳性能的话，我们必须对参数进行一定量的搜索。

注意：本节所说的能够进行搜索的参数均代指超参数。虽然第1章说过会将参数和超参数统称为参数，而且我们确实也能用一些方法来搜索参数，不过因为基于梯度下降法来训练的 `model` 的参数都是用梯度下降来优化的，所以我们能够搜索的只有超参数。

在模型参数搜索方面有许多研究，比较前沿且有效的方法包括贝叶斯优化，可以参见 *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning* (<https://arxiv.org/pdf/1012.2599.pdf>)；强化学习，可以参见 *Neural Architecture Search with Reinforcement Learning* (<https://arxiv.org/pdf/1611.01578.pdf>) 和 *Learning Transferable Architectures for Scalable Image Recognition* (<https://arxiv.org/pdf/1707.07012.pdf>) 这两篇论文。不过，前者需要比较多的数学基础才能理解透彻，而且其算法本身也有超参数需要调整 (trade off between exploration and exploitation)，后者则主要旨在处理非结构化数据 (比如图像、音频、自然语言数据)，而非我们一直以来处理的结构化数据。虽然这两者理论上都能对我们已有模型的参数做出很好的搜索，但是想要进入它们的门槛却并不低。

幸运的是，由于第 5 章所介绍的诸多技术 (WnD、DNDF、Pruning) 已经相当强大，所以即使只采用比较原始的参数搜索方法——随机搜索 (Random Search) 和网格搜索 (Grid Search)，出来的效果就已经相当不错。换句话说，第 5 章所介绍的技术的高性能大大减小了我们需要进行搜索的参数空间，从而即使用朴素的搜索方法也能找到不错的参数。正因此，本节将主要介绍的就是网格搜索和随机搜索的算法及具体实现，对于贝叶斯优化和强化学习则不会涉及，感兴趣的读者可以参见上面提到的三篇论文及相应的参考文献。

网格搜索和随机搜索都属于比较“暴力”的方法，它们拥有相同的本质：用某种参数生成方式来生成一些候选的参数，并通过遍历的方法来找到这些参数中最优的参数。下面是它们共通的算法框架，如算法 7.2 所示。

算法 7.2 (朴素) 参数搜索

输入：某个待评估的模型 G ，评估函数 s ，数据集 X ，多次实验的次数 k ，参数生成方法 P

过程：

(1) 将数据集 X 随机划分为训练用数据集 X_{tr} 与测试用数据集 X_{te} ：

$$X \rightarrow X_{tr} \cup X_{te}, (X_{tr} \cap X_{te} = \emptyset)$$

(2) 利用参数生成方法 P ，生成所有待搜索的参数 p_1, p_2, \dots, p_m

(3) 遍历每个 p_i ($i = 1, 2, \dots, m$) 并利用 s 、 k 和 X_{tr} 、 X_{te} 对参数为 p_i 时的 G 进行多次实验，记录其综合评估 s_i

(4) 选出候选参数中使得 s_i 最大的参数 \hat{p} 作为搜索的结果

输出：参数 \hat{p}

在算法 7.2 的框架下，随机搜索与网格搜索的唯一区别，仅在于参数生成方法 P 的不同。此外通过后文的相应介绍还可以看出，随机搜索可以看作网格搜索的一个“子集”。

与多次实验时的介绍顺序类似，在接下来的第 1 节中我们将会先对参数搜索的整体框架进行展示，然后会在第 2 节中介绍随机搜索和网格搜索，并在第 3 节到第 5 节中分别叙述参数搜索后参数的选取方法、参数搜索的收尾工作以及一套简易但高效的参数搜索方案。

7.3.1 参数搜索的框架

从算法 7.2 可以看出，(朴素) 参数搜索本身是一个比较简单的过程。但是它的难点和多次实验的难点基本一致：我们需要将参数搜索嵌入之前实现的 model 中。而且从算法 7.2 的第(3)

步可以看到，我们还需利用到上一节实现的多次实验来对当前的模型参数做出合理的评估。

考虑到这里面牵扯太多不同情境下实现的功能，如果直接在数据源文件夹中进行相应的工作可能会产生不必要的混乱，所以一种常见的做法是：在运行这种较大规模的、涉及较多交叉功能的程序时，单独地开辟一个“临时数据文件夹”来存储该程序将用到的数据。这种做法一方面能够保证在调试时不会将数据源弄坏，另一方面则能保证这个大规模程序运行时的相对独立性，从而方便我们调试以及进一步的开发。

在这种思想下，在完成算法 7.2 中第 (1) 步——将数据集 X 划分为 X_{tr} 和 X_{te} 后，我们就需要把 X_{tr} 和 X_{te} 保存在一个临时文件夹中。这就能看出这种做法的第三个好处：能够很方便地比较算法之间的优劣。具体而言，由于模型 G 在参数搜索的过程中一直在用 X_{tr} 和 X_{te} 分别作为训练、测试用数据集，而且参数搜索完之后能够得到候选参数中的最优参数 \hat{p} ，所以如果想比较 G 与另外一个算法对应的模型 \tilde{G} 之间的优劣的话，由于 X_{tr} 和 X_{te} 被保存在了一个临时文件夹中，所以就只需在临时文件夹中读取出 X_{tr} 和 X_{te} ，然后让 \tilde{G} 用 X_{tr} 和 X_{te} 分别作为训练、测试用数据集并拿它的表现与 G 的表现进行对比即可。

此外与多次实验类似，我们需要引入单个参数下多次实验的时间限制与整个参数搜索过程的时间限制，这样才能满足广泛实际应用的需求。否则如果不加时间限制的话，假设数据集样本量很大、候选参数很多、模型 G 的训练速度又很慢的话，整个参数搜索过程的耗时可能就会变得难以忍受。

以上就是管理参数搜索时的两个重要思想，参数搜索框架的具体代码实现如下所示。与多次实验类似，我们会将代码分成进入主循环之前和进入主循环之后这两部分来进行说明。首先来看看进入参数搜索主循环之前的实现。

注意：为了简洁，我们会省略一些与输出信息相关的实现并把重点放在核心部分的实现上，对完整实现感兴趣的读者可以参见之前提供过的链接。

```
01     # 能够获取参数搜索当前总耗时的 property
02     # 其中, self._param_search_t 是记录参数搜索开始时的时间属性
03     @property
04     def param_search_time_delta(self):
05         return time.time() - self._param_search_t
06
07     """
08     参数搜索的整体框架
09     params: 候选参数组成的 list
10     search_with_test_set: 是否使用  $X_{te}$  来辅助搜索参数
11     switch_to_best_param: 是否将模型参数设置为最优参数  $\hat{p}$ 
12     single_search_time_limit: 单个参数下多次实验的时间限制
13         默认值为 None, 代表着该时间限制将根据总的时间限制自动生成
14         具体的生成方法会在后文代码注释中给出
15     param_search_time_limit: 参数搜索的总时间限制, 单位为秒
16     k, data, cv_rate, test_rate, sample_weights, kwargs:
17         多次实验中的相应参数
18     """
```

```

19     def param_search(self, params,
20         search with test set=True, switch to best param=True,
21         single search time limit=None, param search time limit=3600,
22         k=3, data=None, cv rate=0.1, test rate=0.1,
23         sample weights=None, **kwargs
24     ):
25         # 记录参数搜索开始的时间
26         self.param_search_t = time.time()
27         # 记录参数搜索的总耗时
28         self.param_search_time_limit = param_search_time_limit
29         # 获取参数搜索时的 logger
30         logger = self.param_search_logger
31         # 将标志着“是否正在搜索参数”的属性设置为 True
32         self.searching_params = True
33         # 定义一个“超参数基底”，今后在搜索参数时，我们就会
34         # 以这个“基底”为基础进行拓展
35         self.settings_base = {
36             "model param settings": deepcopy(
37                 self.model_param_settings),
38             "model structure settings": deepcopy(
39                 self.model_structure_settings)
40         }
41         # 初始化记录参数搜索过程中各个参数下模型表现的属性
42         self.mean_record, self.std_record = [], []
43         # 调用相应方法，将数据集  $x$  划分为 $X_{tr}$ 和 $X_{te}$ 
44         # 需要指出的是，该方法还会创建一个临时数据文件夹
45         # 并把 $X_{tr}$ 和 $X_{te}$ 保存在这个临时数据文件夹中
46         self.prepare_param_search_data(data, test_rate)
47         # 获取候选参数的个数
48         n_param = len(params)

```

其中，代码 46 行处调用的 `self.prepare_param_search_data` 方法的相关实现如下所示。

```

01     # 定义一个能够返回临时数据文件夹的 property
02     @property
03     def data_cache_folder_name(self):
04         # 这里我们将临时数据文件夹设置为“./ Tmp/ Cache”
05         # 大家完全可以根据需求来更改该目录
06         folder = os.path.join(os.getcwd(), " Tmp", " Cache")
07         # 如果该数据文件夹不存在，就创建它
08         if not os.path.isdir(folder):
09             os.makedirs(folder)
10         return folder
11
12     def prepare_param_search_data(self, data, test_rate):
13         # 由于我们会默认用 txt 文件来记录 $X_{tr}$ 和 $X_{te}$ 
14         # 所以要将 self.data info 中的“file type”一项默认设置为“txt”
15         file_type = self.data_info.setdefault("file type", "txt")
16         # 同理，将 self.data info 中的“data folder”一项默认设置为“Data”
17         data_folder = self.data_info.setdefault("data_folder", "_Data")

```

```

18     # 用相应的属性记录下当前的 file type 和 data folder
19     self.file_type_store = file_type
20     self.data_folder_store = data_folder
21     # 如果用户已经提供了 $X_{tr}$ 和 $X_{te}$ 的话, 就无须进一步的操作了
22     if data is not None:
23         return data
24     # 否则, 就需要创建一个临时数据文件夹来处理参数搜索的相关事宜
25     # 首先, 根据 data folder 和 self.name 属性, 获取文件数据集的名字
26     target = os.path.join(data_folder, self.name)
27     # 然后, 利用上一章实现的 self.get_data_from_file 方法
28     # 从文件中获取数据集
29     data, test_rate = self.get_data_from_file(file_type, test_rate, target)
30     # 如果文件数据集已经分好了 $X_{tr}$ 和 $X_{te}$ 的话, 就直接进行相应的赋值
31     if isinstance(data, tuple):
32         train_data, test_data = data
33     # 否则, 根据 test_rate 来随机划分出 $X_{tr}$ 和 $X_{te}$ 
34     else:
35         if test_rate > 0:
36             random.shuffle(data)
37             n_train = int(len(data) * (1 - test_rate))
38             train_data, test_data = data[:n_train], data[n_train:]
39         else:
40             train_data, test_data = data, None
41     # 继而调用相应的 property 来获得临时数据文件夹的路径
42     cache_folder = self.data_cache_folder_name
43     # 并利用该路径和 self.name 属性来获取当前临时文件数据集
44     # 所在的文件夹的名字
45     cache_target = os.path.join(cache_folder, self.name)
46     # 如果该文件夹不存在, 就创建它
47     if not os.path.isdir(cache_target):
48         os.makedirs(cache_target)
49     # 将 $X_{tr}$ 和 $X_{te}$ 分别写进相应的(临时)txt文件中
50     with open(os.path.join(cache_target, "train.txt"), "w") as file:
51         file.write("\n".join([" ".join(line) for line in train_data]))
52     if test_data is not None:
53         with open(os.path.join(cache_target, "test.txt"), "w") as file:
54             file.write("\n".join([" ".join(line) for line in test_data]))
55     # 将 file_type 一项的值改为 txt, 因为上述临时数据文件的后缀是 txt
56     self.data_info["file_type"] = "txt"
57     # 同样的, 将 data_folder 一项的值改为临时数据文件夹
58     self.data_info["data_folder"] = cache_folder

```

在上述实现下, 如果我们在某个程序中运行参数搜索的话, 就会和多次实验一样, 在该程序所在的路径中创建一个“_Tmp”文件夹。而这里与多次实验的不同之处在于, 多次实验是在“_Tmp”中创建了一个叫“_Logging”的文件夹以存储日志文件, 而上述实现则会在“_Tmp”中创建一个叫“_Cache”的文件夹以作为临时数据文件夹。

注意: 参数搜索产生的日志文件仍然会保存在“_Logging”文件夹中。

以上便是主循环之前的实现，下面我们来看看参数搜索整体框架主循环的实现。

```

49     # 遍历候选参数
50     for i, param in enumerate(params):
51         # 调用相应方法，初始化当前所用的计算图 Graph
52         # 从而使得各个参数下的多次实验不会相互影响
53         self.reset_graph(i)
54         # 调用相应方法，向日志文件写入当前的参数设置
55         # 该方法的实现用了一些 Python 技巧且与参数搜索本身关系不大
56         # 所以我们略去其详细的说明
57         self._log_param_msg(i, param)
58         # 调用相应方法，将当前模型 G 的参数设置为 param
59         self.update_param(param)
60         # 计算出剩余可用的时间
61         time_left = param_search_time_limit - self.param_search_time_delta
62         # 如果单个参数下多次实验的时间限制为 None 的话
63         # 就根据剩余时间和剩余参数数量计算出当前的时间限制
64         if single_search_time_limit is None:
65             local_time_limit = time_left / (n_param - i)
66         else:
67             local_time_limit = single_search_time_limit
68         # 将参数中 time_limit 一项的值设置为相应的时间限制
69         kwargs["time_limit"] = min(local_time_limit, time_left)
70         # 调用 self.k_random 方法进行多次实验
71         # 如果该方法没有返回 None，说明尚有时间剩余
72         if self.k_random(
73             k, data, cv_rate, test_rate, sample_weights, **kwargs
74         ) is not None:
75             # 此时就将训练好的模型进行保存以便复用
76             self.save()
77             # 同时，记录下模型相应的综合评估
78             self.mean_record.append(self.k_performances_mean)
79             self.std_record.append(self.k_performances_std)
80             # 如果当前已经超出了时间限制的话，就退出参数搜索
81             if self.param_search_time_delta >= param_search_time_limit:
82                 break
83         # 调用相应方法选出最佳参数  $\hat{p}$  (best_param)
84         # 同时一并获取  $\hat{p}$  是“第几个” (best_idx) 参数 (即其下标)
85         best_idx, best_param = self.select_param(
86             params, search_with_test_set)
87         # 利用相应方法，将最佳参数设置写入日志文件
88         self._log_param_msg(-1, best_param)
89         # 将参数搜索的结果信息写入日志文件
90         # 这些信息的具体形式会在后文进行展示
91         msg = ""
92         for i, (mean, std) in enumerate(zip(self.mean_record, self.std_record)):
93             msg += "  -{} Mean   | ".format(">" if i == best_idx else " ")
94             msg += self._print_metrics(
95                 self._metric_name, *mean, only_return=True) + "\n"

```

```

96         msg += " -{} Std | ".format(">" if i == best_idx else " ")
97         msg += self.print_metrics(self.metric_name, *std, only_return=True)
98         if i != len(self.mean_record) - 1:
99             msg += "\n" + "-" * 100 + "\n"
100     self.log_block(msg(
101         "Generating performances", body=msg,
102         level=logging.DEBUG, logger=logger)
103     # 如果需要将模型参数设置为最优参数p的话
104     if switch_to_best_param:
105         # 就先重置当前的TensorFlow运算图
106         self.reset_graph(-1)
107         # 并调用相应方法进行参数设置
108         self.update_param(best_param)
109     # 调用相应方法进行收尾工作
110     self.param_search_completion()
111     return self

```

在这种实现下，参数搜索中写入日志文件的和模型评估相关的信息将大致如图 7.8 和图 7.9 所示。

```

Performance of run 1 | auc - Train : 0.99737461 CV : 0.99713221 Test : 0.99681568
Performance of run 2 | auc - Train : 0.99941558 CV : 0.99028213 Test : 0.99718310
Performance of run 3 | auc - Train : 0.99910613 CV : 0.99713221 Test : 0.99755052
Generating performance summary
=====
Result
-----
- Mean | auc - Train : 0.99863210 CV : 0.99484885 Test : 0.99718310
- Std | auc - Train : 0.00089812 CV : 0.00322916 Test : 0.00030000
-----

```

图 7.8 参数搜索中单个参数下的评估信息（节选）

```

-----
2018-01-19 14:39:10 - 38_param_search - DEBUG - Generating performances
=====
Result
-----
-> Mean | auc - Train : 0.99863210 CV : 0.99484885 Test : 0.99718310
-> Std | auc - Train : 0.00089812 CV : 0.00322916 Test : 0.00030000
-----
- Mean | auc - Train : 0.98994473 CV : 0.99677317 Test : 0.99506022
- Std | auc - Train : 0.01324941 CV : 0.00110429 Test : 0.00207925
-----
- Mean | auc - Train : 0.99483081 CV : 0.97894234 Test : 0.99485609
- Std | auc - Train : 0.00199824 CV : 0.01569378 Test : 0.00062450
-----
- Mean | auc - Train : 0.99634834 CV : 0.98281918 Test : 0.99546846
- Std | auc - Train : 0.00071209 CV : 0.00408976 Test : 0.00098488
-----
- Mean | auc - Train : 0.99674201 CV : 0.99084136 Test : 0.99624413
- Std | auc - Train : 0.00006345 CV : 0.00475981 Test : 0.00075055
-----
- Mean | auc - Train : 0.99535502 CV : 0.98860034 Test : 0.99503980
- Std | auc - Train : 0.00035885 CV : 0.0016875 Test : 0.0006134
-----

```

图 7.9 参数搜索的评估信息汇总（节选）

可以看到，我们不仅记录下了单个参数里的多次实验中每次实验的模型表现，也记录下了多次实验下模型的综合表现（这其实是上一节中多次实验的相应实现所保证的）。同时，在参数搜索结束后，我们还会统一罗列出每个参数下模型的综合表现，并会在最优参数对应的模型表现前面用特殊的符号（“->”）进行标识，这样就能让我们方便地进行后续操作。

此外，在上述实现中我们用到了各式各样的方法，其中第 85 行调用的 `self._select_param` 方法将会在 7.3.4 节进行说明，第 110 行调用的 `self._param_search_completion` 方法的说明将会分别在 7.3.4 节和 7.3.5 节中给出，第 59 行、第 108 行调用的 `self._update_param` 方法的实现则如下所示。

```

112     def update_param(self, param):
113         # 利用 self._settings_base 属性
114         # 将所有和参数设置相关的属性初始化
115         self.model_built = False
116         self.settings_initialized = False
117         self.model_param_settings = deepcopy(
118             self.settings_base["model_param_settings"])
119         self.model_structure_settings = deepcopy(
120             self.settings_base["model_structure_settings"])
121         # 获取新的参数设置
122         new_model_param_settings = param.get(
123             "model_param_settings", {})
124         new_model_structure_settings = param.get(
125             "model_structure_settings", {})
126         # 根据新的参数设置，在“超参数基底”的基础上更新参数
127         self.model_param_settings.update(new_model_param_settings)
128         self.model_structure_settings.update(new_model_structure_settings)
129         # 根据当前的参数设置，逐一处理
130         # 软剪枝、DNDF、缺失值处理器、预处理器等技术对应的属性
131         if not self.model_structure_settings.get("use_pruner", True):
132             self.pruner = None
133         if not self.model_structure_settings.get("use_dndf", True):
134             self.dndf = None
135         if not self.model_structure_settings.get("use_dndf_pruner", False):
136             self.dndf_pruner = None
137         if self.nan_handler is not None:
138             self.nan_handler.reset()
139         if self.pre_processors:
140             self._pre_processors = {}

```

而第 106 行调用的 `self.reset_graph` 方法的实现则如下所示。

```

141     def reset_graph(self, i):
142         # 先将已有的运算图完全删掉
143         del self._graph
144         # 并把已有的 Session 置为 None
145         self._sess = None
146         # 然后再初始化一张运算图
147         self._graph = tf.Graph()
148         # 并更新记录着当前正在搜索“第几个”参数的属性
149         self._search_cursor = i

```

以上就是参数搜索整体框架的全部实现，除去为了兼容 `model` 而实现的相应方法（比如 `self._update_param` 方法和 `self.reset_graph` 方法的实现）以外，其余部分的实现基本和算法 7.2

有一一对应的关系。

在本节的最后，我们来补上 7.2.1 节中用到过的但却没有展开说明的方法——self._handle_param_search_time_limit 方法——的具体代码。

```

150     def handle_param_search_time_limit(self, time_limit):
151         # 如果当前不是处于参数搜索的状态的话
152         if self.param_search_time_limit is None:
153             # 直接在时间限制上减去当前多次实验总耗时即可
154             time_limit -= self.k_series_time_delta
155         else:
156             # 否则，就要考虑上参数搜索的时间限制
157             time_limit = min(
158                 time_limit,
159                 self.param_search_time_limit - self.k_series_time_delta
160             )
161         # 因为上面这些代码已经在时间限制中减去了当前多次实验的总耗时
162         # 所以我们要重置多次实验的开始时间
163         self.k_series_t = time.time()
164         # 并返回当前“真正的”时间限制
165         return time_limit

```

7.3.2* 随机搜索与网格搜索

7.3.1 节我们介绍了参数搜索的框架，这一节我们来介绍两种朴素但在 AdvancedNN 上有效的参数搜索算法：随机搜索和网格搜索。

正如本节开头所说，参数搜索和网格搜索的唯一不同仅在于算法 7.2 中的参数生成方法 P 的不同。直观上来说，网格搜索比较“中规中矩”，它会遍历所有给定的可能性来进行搜索；而随机搜索则会更“率性”一些，它只会在给定的可能性中随机地挑选出若干个参数来进行搜索。

接下来我们就更为严谨地叙述一下这两种算法，为此需要先定义一些符号。假设现在总共有 p 个离散型参数 ($d_1 \sim d_p$) 和 q 个连续型参数 ($c_1 \sim c_q$) 需要进行搜索，再不妨假设第 i 个离散型参数总共有 n_i 个取值，且

$$d_i \in \{d_i^{(1)}, d_i^{(2)}, \dots, d_i^{(n_i)}\}, \quad \forall i = 1, 2, \dots, p$$

以及不妨设第 j 个连续型参数的取值范围为 $(c_j^{(l)}, c_j^{(u)})$ ，即

$$c_j^{(l)} \leq c_j < c_j^{(u)}, \quad \forall j = 1, 2, \dots, q$$

那么，最完整的网格搜索算法就是遍历所有 d_i 的可能取值，以及按照一定的规律对所有 c_j 采样，并在这些参数组合中选出最优的参数组合。具体而言，假设我们统一对每个连续型参数 c_j 采 m_j 个等间隔的样本，即把第 j 个连续型参数看成如下的离散型参数：

$$\tilde{c}_j \in \{\tilde{c}_j^{(1)}, \tilde{c}_j^{(2)}, \dots, \tilde{c}_j^{(m_j)}\}, \quad \forall j = 1, 2, \dots, q$$

其中

$$\tilde{c}_j^{(k)} = c_j^{(l)} + \frac{k-1}{m_j} \cdot \Delta c_j, \quad \forall k = 1, 2, \dots, m_j$$

且

$$\Delta c_j = c_j^{(u)} - c_j^{(l)}$$

的话，那么网格搜索生成的候选参数集合 P_{grid} 就是

$$P_{\text{grid}} = \left\{ \left\{ d_1^{(s_1)}, \dots, d_p^{(s_p)}, \tilde{c}_1^{(t_1)}, \dots, \tilde{c}_1^{(t_q)} \right\} \mid s_i \in \{1, 2, \dots, n_i\}, t_j \in \{1, 2, \dots, m_j\} \right\}$$

由此可以看出，网格搜索虽然确实非常全面，但它的运行开销在大多数情况下也是难以忍受的，因为它总共需要进行

$$\prod_{i=1}^p n_i \prod_{j=1}^q m_j$$

次参数的搜索，这对于参数的数量和参数的取值个数而言都是非线性的增长。此外，网格搜索的这种全面性在很多情况下也并非好事：如果某个参数取值（比如， $d_i = d_i^{(1)}$ ）是非常差的选择的话，那么无论其他参数再怎么好，可能也无法弥补这单个极差的参数所带来的性能损伤。在这种情况下，我们本应在几次实验之后就发现这点并把剩余包含 $d_i = d_i^{(1)}$ 的参数组合都去掉，但网格搜索却会把所有包含 $d_i = d_i^{(1)}$ 的参数组合的实验都做完，从而带来很大的浪费。

为了解决这种冗余性，随机搜索的思想就应运而生了。事实上正如本节开头所说，随机搜索可以被看作网格搜索的一个子集，因为它的候选参数集 P_{rand} 就是网格搜索的候选参数集 P_{grid} 的子集。具体而言，假设我们预先设置好搜索 n_{rand} 个参数的话，就只需让 P_{rand} 满足：

$$(P_{\text{rand}} \subset P_{\text{grid}}) \wedge (|P_{\text{rand}}| = n_{\text{rand}})$$

即可。从实现的角度来看的话，只需把 P_{grid} 对应的 list 打乱，然后取前 n_{rand} 个即可。

以上就是网格搜索和随机搜索算法的叙述。虽然它们并不难，但由于我们在前面的章节中统一使用了字典来作为参数的管理媒介，所以实际的实现却没那么简单，这也是为何本节的标题处带了星号(*)的原因。接下来将要展示的随机搜索和网格搜索的实现将会用到大量的 Python 技巧，它们就是本节的额外知识点。如果只对参数搜索算法本身感兴趣的话，不看接下来的实现也可以；不过如果确实想要知道它们相应的、基于 model 的、较为优雅的实现的话，可以接着往下看。

在展示具体的代码之前，我们先来说说实现的逻辑。由于在此前实现的 model 中，我们使用了 model_param_settings、model_structure_settings、data_info 等字典（以后统一简称它们为“超参数字典”）来管理超参数（比如，上述三个字典就分别存储着模型超参数、结构超参数与数据超参数），所以我们在枚举需要进行搜索的参数时就会有两种枚举方法：

- 对每个超参数字典进行枚举（list_first）。
- 在每个超参数字典内部进行枚举（dict_first）。

其中，第一种枚举方法简单易懂，只需把所有需要枚举的参数都打包成参数组合并塞进参数搜索框架即可；但是，当需要搜索的参数个数有很多，且真正需要做网格搜索时，第一种枚举方法会显得比较累赘，此时我们就需要用第二种枚举方法来枚举。比如，现在有 a_1 和 a_2 两个模型超参数和 $b_1 \sim b_3$ 三个结构超参数，它们都分别只有两个参数取值。如果想要对这5个参数做网格搜索的话，第一种枚举方法就需要写成如下所示的代码。

```
01 grid_params = {
02     "model_param_settings": [
03         {"a1": a11, "a2": a21},
04         {"a1": a11, "a2": a22},
05         {"a1": a12, "a2": a21},
06         {"a1": a12, "a2": a22},
07     ],
08     "model_structure_settings": [
09         {"b1": b11, "b2": b21, "b3": b31},
10         {"b1": b11, "b2": b21, "b3": b32},
11         {"b1": b11, "b2": b22, "b3": b31},
12         {"b1": b11, "b2": b22, "b3": b32},
13         {"b1": b12, "b2": b21, "b3": b31},
14         {"b1": b12, "b2": b21, "b3": b32},
15         {"b1": b12, "b2": b22, "b3": b31},
16         {"b1": b12, "b2": b22, "b3": b32},
17     ]
18 }
19
```

即第一种枚举方法要进行大量的重复书写，这在多数情况下是非常不优雅的。而如果使用第二种枚举方法的话，就只需写成如下所示的代码。

```
01 grid_params = {
02     "model_param_settings": {
03         "a1": [a11, a12], "a2": [a21, a22]
04     },
05     "model_structure_settings": {
06         "b1": [b11, b12], "b2": [b21, b22], "b3": [b31, b32]
07     }
08 }
```

不过相对应的，第一种枚举方法的实现相对来说比较简易，而第二种枚举方法的实现就需要不少技巧了，下面我们来看看它们各自在随机搜索下的代码。

```
01 """
02     随机搜索的实现
03     n: 搜索的参数数量n_rand
04     如果是-1的话就意味着全搜索（即网格搜索）
05     grid_params: 上面展示过的两种枚举方法代码中的 grid_params
06     grid_order: list_first 代表第一种枚举方法，dict_first 代表第二种
07     之后的参数则都是参数搜索框架中的参数，这里不再赘述
08 """
```

```

09     def random_search(self, n, grid_params, grid_order="list first",
10                       search_with_test_set=True, switch_to_best_params=True,
11                       single_search_time_limit=None, param_search_time_limit=3600,
12                       k=3, data=None, cv_rate=0.1, test_rate=0.1,
13                       sample_weights=None, **kwargs
14     ):
15         # 第一种枚举方法的实现
16         if grid_order == "list first":
17             # 先根据名字来获取各个超参数字典的处理顺序
18             param_types = sorted(grid_params)
19             # 然后获取各个超参数字典中的参数组合个数
20             # 并根据组合个数来生成各个参数组合所对应的下标
21             n_param_base = [
22                 np.arange(len(grid_params[param_type]))
23                 for param_type in param_types
24             ]
25             # 最后生成候选参数列表
26             # 这里我们用到了 Python 自带标准库 itertools 中的 product 方法
27             # 来生成各个超参数字典的参数组合的笛卡儿积
28             # 其用法可参见:
29             # https://docs.python.org/3.6/library/itertools.html#itertools.product
30             params = [
31                 {
32                     param_type: grid_params[param_type][indices[i]]
33                     for i, param_type in enumerate(param_types)
34                 } for indices in itertools.product(*n_param_base)
35             ]
36         # 第二种枚举方法的实现
37         elif grid_order == "dict first":
38             # 根据名字来获取各个超参数字典的处理顺序
39             param_types = sorted(grid_params)
40             # 获取各个超参数字典中的超参数名字, 同时进行排序
41             params_names = [
42                 sorted(grid_params[param_type])
43                 for param_type in param_types
44             ]
45             # 获取各个超参数字典中的超参数个数并进行累计求和
46             params_names_cumsum = np.cumsum([0] + [
47                 len(params_name) for params_name in params_names
48             ])
49             # 获取各个超参数的候选参数个数
50             # 并根据候选参数个数来生成各个候选参数对应的下标
51             n_param_base = sum([
52                 np.arange(len(grid_params[param_type][param_name]))
53                 for param_name in params_name
54             ] for param_type, params_name in zip(
55                 param_types, params_names)), []
56         )
57         # 生成候选参数列表

```

```

58     params = [
59         {
60             param type: {
61                 local params: grid params[
62                     param type][local params][indices[cumsum + j]
63                 ] for j, local params in enumerate(params names[i])
64             } for i, (param type, cumsum) in enumerate(
65                 zip(param types, params names cumsum))
66         } for indices in itertools.product(*n param base)
67     ]
68     # 如果 $n_{\text{rand}}$ 对应的参数n大于0,说明要进行随机搜索
69     # 所以就随机挑选出n个参数作为候选参数
70     if n > 0:
71         params = [
72             params[i] for i in np.random.permutation(len(params))[:n]]
73     # 调用上一节实现的 param search 方法进行参数搜索
74     return self.param search(
75         params,
76         search with test set, switch to best params,
77         single search time limit, param search time limit,
78         k, data, cv rate, test rate, sample weights, **kwargs
79     )

```

上述代码中生成候选参数列表的部分是比较难理解的,其余部分则相对比较常规。注意,该随机搜索的实现是兼容网格搜索的,所以在此基础上的网格搜索的实现非常简洁。

```

80     def grid_search(self, grid_params, grid_order="list_first",
81         search_with_test_set=True, switch_to_best_params=True,
82         single_search_time_limit=None, param_search_time_limit=3600,
83         k=3, data=None, cv_rate=0.1, test_rate=0.1,
84         sample_weights=None, **kwargs
85     ):
86         # 只需将 random_search 中的参数 n 设置为-1, 其实就是网格搜索了
87         return self.random_search(
88             -1, grid_params, grid_order,
89             search_with_test_set, switch_to_best_params,
90             single_search_time_limit, param_search_time_limit,
91             k, data, cv_rate, test_rate, sample_weights, **kwargs
92         )

```

最后需要指出的是,虽然随机搜索确实总能看作网格搜索的子集,但是在一般情况下,连续型参数的随机搜索都特指网格粒度接近最细时的网格搜索的子集。比如,如果我想对隐藏层神经元个数做随机搜索的话,由于神经元个数必须是整数,所以最细网格粒度就是 1,这意味着随机搜索时会以 1 为网格粒度来从网格搜索中随机选出一个子集。具体而言,假设我们设定神经元个数最小为 16、最大为 128 的话,随机搜索就会从 16 到 128 之间随机挑选出若干个整数来进行搜索。再比如,我想对学习速率进行搜索,且设定学习速率最小为 0.001 最大为 0.1 的话,那么随机搜索一般就是指从 0.001 到 0.1 之间任意挑选出若干个数值来搜索,而不是非得像网格搜索那样先将 0.001 到 0.1 这个大区间分成若干子区间,然后取子区间的端点作为候选参数。

总而言之，一般意义下的随机搜索在搜索连续型特征时会“更自由”，而不一定要像网格搜索那样先将连续型特征离散化。所以我们此前一直将随机搜索视为网格搜索的子集，一方面是因为从极限的角度来看确实如此，另一方面则是为了实现上的便利。

不过，贪图便利而不将功能实现完整显然是不负责任的行为，所以笔者也将这种更自由的随机搜索进行了实现。由于相应的实现和上述 `random_search` 方法的实现大同小异，所以就不在此赘述了，感兴趣的读者可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/DistBase.py 中 `DistMixin` 的 `range_search` 方法。

7.3.3 参数的选取

前两节我们介绍了如何遍历候选参数并得到各个参数下模型的综合表现，这一节我们则会叙述一种参数的选取方法。一种自然的想法是比较各个参数下模型性能的平均值，并根据性能平均值的大小来决定选取哪个参数，但是这种做法会遇到两个问题：

- 我们现在获得了训练集、交叉验证集和测试集上模型的性能平均值，它们的“重要性”应该如何分配？
- 如果只考虑平均值的话就会忽略模型的稳定性，应该如何把稳定性也考虑进来呢？

对这两个问题的解答就构成了参数选取的逻辑。下面就来介绍诸多解决方案的其中一套，它并不一定是最好的，但却是在大多数情况下行之有效的：

- 如果只看训练集和交叉验证集的话，就按照5:95的权重进行配比。
- 如果同时看训练集、交叉验证集和测试集的话，就按照5:10:85的权重进行配比。
- 由于我们还记录了各个模型下模型性能的标准差，考虑到标准差是较好的模型稳定性的衡量标准（标准差越低表示模型越稳定），所以我们将模型的最终“分数”设置为性能均值减去性能标准差即可。

在这种思想下，参数选取的相应实现如下所示。

```
01 # 参数 params 代指所有的候选参数
02 # 参数 search_with_test_set 代指是否使用测试集来辅助选取参数
03 def select_param(self, params, search_with_test_set):
04     # 定义一个 list 来记录所有候选参数下模型的“分数”
05     scores = []
06     # 看看当前模型的评估标准是越大越好还是越小越好
07     sign = Metrics.sign_dict[self.metric_name]
08     # 遍历每个参数下模型性能的平均值和标准差
09     for mean, std in zip(self.mean_record, self.std_record):
10         # 如果没有提供过测试集或选择了不用测试集的话
11         if len(mean) == 2 or not search_with_test_set:
12             # 就根据5:95的比例，算出加权均值与加权标准差
13             train_mean, cv_mean = mean
14             train_std, cv_std = std
15             weighted_mean = 0.05 * train_mean + 0.95 * cv_mean
16             weighted_std = 0.05 * train_std + 0.95 * cv_std
```

```

17         # 否则, 就根据5:10:85的比例来计算
18         else:
19             train mean, cv mean, test mean = mean
20             train std, cv std, test std = std
21             weighted mean = (
22                 0.05 * train mean + 0.1 * cv mean + 0.85 * test mean)
23             weighted std = (
24                 0.05 * train std + 0.1 * cv std + 0.85 * test std)
25             # 调用 self. get score 方法来获取模型的“分数”
26             # 该方法的实现马上进行说明
27             scores.append(self. get score(weighted mean, weighted std, sign))
28             # 处理分数中的非法值(把所有的 nan 换成负无穷)
29             scores = np.array(scores, np.float32)
30             scores[np.isnan(scores)] = -np.inf
31             # 利用 numpy 中的 argmax 方法获取最优参数及其下标并返回
32             best idx = np.argmax(scores)
33             return best idx, params[best idx]
34
35     @staticmethod
36     def get score(mean, std, sign):
37         # 如果当前模型的评估指标是越大越好的话
38         if sign > 0:
39             # 就返回均值减标准差
40             return mean - std
41         # 否则, 就返回均值加标准差
42         return mean + std

```

以上就是参数选取的相应实现, 接下来我们只需把收尾工作做好即可。

7.3.4 参数搜索的收尾工作

参数搜索的收尾工作相对而言要简单一些, 因为绝大多数的收尾工作都在单个参数下模型的多次实验中做完了, 参数搜索本身的收尾工作就只需要恢复一些模型属性即可, 代码如下所示。

```

01     def param search completion(self):
02         # 将标志着“是否正在搜索参数”的属性设置为 False
03         self. searching params = False
04         # 将记录参数搜索时间限制的属性重置为 None
05         self.param search time limit = None
06         # 将标志着“是否已经初始化数据超参数”的属性设置为 False
07         self. data info initialized = False
08         # 将数据超参数的 file_type 和 data_folder 恢复如初
09         self.data info["file type"] = self. file type store
10         self.data _info["data_folder"] = self._data_folder_store

```

7.3.5 具体的搜索方案

前4节我们已经将所有参数搜索的相关实现说明了一遍, 在本节中, 我们就来看一套具体

可行的参数搜索方案。虽然我们前面说网格搜索可能会带来冗余，但是如果能够保证各项候选参数的有效性的话，网格搜索是比较好的选择。特别是我们在第 5 章介绍的各项技术，都相当有效且各有各的优点，所以对它们进行网格搜索就是非常不错的搜索方案，其相应的实现如下所示。

```

01     # 参数 level 代指参数搜索的级别，级别越高代指搜索的参数越多
02     # 其余参数均为参数搜索框架的参数，这里不再赘述
03     def empirical_search(self,
04         search with test set=True, switch to best params=True,
05         level=3, single search time limit=None, param search time limit=3600,
06         k=3, data=None, cv rate=0.1, test rate=0.1,
07         sample weights=None, **kwargs
08     ):
09         # 定义基础的候选参数
10         # 具体而言，我们至少需要搜索如下 3 组结构超参数：
11         # 1) 朴素的神经网络，即不使用 WnD、DNDF 和剪枝算法
12         # 2) 在 1) 的基础上，使用软剪枝算法
13         # 3) 在 2) 的基础上，使用 WnD 和 DNDF，但不在 DNDF 中使用剪枝
14         grid_params = {
15             "model structure settings": [
16                 {"use wide network": False, "use pruner": False,
17                  "use dndf pruner": False},
18                 {"use wide network": False, "use pruner": True,
19                  "use dndf pruner": False},
20                 {"use wide network": True, "use pruner": True,
21                  "use dndf pruner": False},
22             ]
23         }
24         # 如果搜索级别不小于 2 的话，就加上如下 3 组结构超参数：
25         # 4) 在上述 3) 的基础上，在 DNDF 中使用剪枝
26         # 5) 在 4) 的基础上，弃用剪枝算法
27         # 6) 在 5) 的基础上，弃用 DNDF 中的剪枝算法
28         if level >= 2:
29             grid_params["model structure settings"] += [
30                 {"use wide network": True, "use pruner": True,
31                  "use dndf pruner": True},
32                 {"use wide network": True, "use pruner": False,
33                  "use dndf pruner": True},
34                 {"use wide network": True, "use pruner": False,
35                  "use dndf pruner": False}
36             ]
37         # 如果搜索级别不小于 3 的话，就加上如下 2 组数据预处理参数
38         # 1) 计算各个特征的均值与标准差时，复用训练集上的均值与标准差
39         # 2) 不复用训练集的均值与标准差
40         if level >= 3:
41             grid_params["pre process settings"] = [
42                 {"reuse_mean_and_std": False},
43                 {"reuse_mean_and_std": True}
44             ]

```

```

45     # 如果搜索级别不小于 4 的话, 就加上如下 2 组模型超参数
46     # 1) 不使用 Batch Normalization 技术
47     # 2) 使用 Batch Normalization 技术
48     if level >= 4:
49         grid_params["model param settings"] = [
50             {"use_batch_norm": False},
51             {"use_batch_norm": True}
52         ]
53     # 如果搜索级别不小于 5 的话, 就加上如下 6 组模型超参数
54     # 它们只是使用或不使用 BN 以及 Mini Batch 大小的枚举
55     # 这里就不一一赘述了
56     if level >= 5:
57         grid_params["model param settings"] = [
58             {"use_batch_norm": False, "batch_size": 64},
59             {"use_batch_norm": False, "batch_size": 128},
60             {"use_batch_norm": False, "batch_size": 256},
61             {"use_batch_norm": True, "batch_size": 64},
62             {"use_batch_norm": True, "batch_size": 128},
63             {"use_batch_norm": True, "batch_size": 256}
64         ]
65     # 调用网格搜索方法进行参数的搜索
66     return self.grid_search(
67         grid_params, "list first",
68         search with test set, switch to best params,
69         single search time limit, param search time limit,
70         k, data, cv_rate, test_rate, sample_weights, **kwargs
71     )

```

从经验上来说的话, 使用第 3 级别的参数搜索就能获得很好的结果了, 我们会在下一节给出其具体的应用方式以及实际表现。

7.4 DistAdvanced 的性能

至此我们已经花费了大量的力气来实现半自动化机器学习框架 **AutoBase** (上一章) 和工程化机器学习框架 **DistMixin** (本章), 但却一直没有用数据集来验证它们的有效性。此外, 在之前的章节中, 我们做实验时用的数据集要么是人造的数据集 (**Madelon**、**异或**、**九宫格**、**噪声数据集**等), 要么是比较简单的数据集 (**mushroom**), 而且普遍也没有和其他模型做对比。所以在这一节中, 我们将有针对性地挑选一些真实的数据集, 并和一个非常有名的自动机器学习第三方库——**AutoSklearn** 进行对比, 从而让大家对第 5 章实现的增强版神经网络 (**AdvancedNN**)、半自动化机器学习框架 **AutoBase** 与工程化机器学习框架 **DistMixin** 的强大有一个更直观的认知。不过正因为我们要和 **AutoSklearn** 进行对比, 所以在进行具体的实验之前, 有必要对 **AutoSklearn** 做一个简要的介绍。

AutoSklearn 是 **AutoML** (<http://automl.chalearn.org/data>) 自动机器学习竞赛冠军得主发明的算法, 其原论文发表在了 **NIPS 2015** 上, 感兴趣的读者可以参见 <http://papers.nips.cc/paper/5872->

efficient-and-robust-automated-machine-learning.pdf。它是基于 scikit-learn 的一个自动化机器学习框架，它和我们在第 6 章实现的半自动化机器学习框架 AutoBase 有着较为本质的不同，具体如下所述。

- 虽然 AutoSklern 可以进行自动化的数据预处理，但它要求我们至少给它提供一个数值型的数据。而对于 AutoBase 而言，我们无须把数值型数据提供给它，而可以只给它提供一个文件路径，然后 AutoBase 就能自动完成文件格式数据的读取、转换与预处理等工作。
- 虽然 AutoBase 能自动化几乎所有的数据准备工作，但是它在单次训练中只能应用在单一的模型（model）上，而且也无法对 model 的参数、超参数进行自动化的调优。而对于 AutoSklern 而言，其最大亮点就在于它不仅能够帮助你自动地从 scikit-learn 的诸多模型中挑选出一个最好的模型，而且还能对这个模型进行自动化的参数调优。

总之可以这么说：AutoSklern 主打自动化模型选择，而 AutoBase 则主打自动化数据准备。需要指出的是，之所以我们实现的 AutoBase 只考虑了数据准备，是因为我们对第 5 章介绍的诸多技术比较有自信，而且参数搜索的部分其实是被本章介绍的 DistMixin 所涵盖的。所以当我们结合了 AdvancedNN、AutoBase 与 DistMixin 之后，得到的工程化神经网络模型（DistAdvanced）就已经是一个自动机器学习模型了。不过与半自动化机器学习框架类似，为了真正应用上工程化机器学习框架，我们需要实现一个沟通 DistMixin 与 model 之间的桥梁——DistMeta：

```

01 class DistMeta(type):
02     def new (mcs, *args, **kwargs):
03         name , bases, attr = args[:3]
04         model, dist mixin = bases
05
06     def init (self, name=None, data info=None,
07             model param settings=None, model structure settings=None,
08             pre process settings=None, nan handler settings=None
09     ):
10         # 初始化一些参数搜索需要用到的属性
11         self. search cursor = None
12         self. param search t = None
13         self.param search time limit = None
14         self.mean record = self.std record = None
15         self. searching params = self. settings base = None
16
17         # 依次调用 DistMixin 和 model 的初始化方法
18         dist mixin. init (self)
19         model. init (
20             self, name, data info,
21             model param settings, model structure settings,
22             pre process settings, nan handler settings
23         )
24
25     attr[" init "] = init
26     return type(name_, bases, attr)

```


可以看到，由于工程化机器学习框架的本质是 Mixin，所以它和 model 之间的兼容性会比上一章实现的 AutoBase 和 model 的兼容性要更好，所以在 DistMeta 中只需把 DistMixin 中会用到的属性进行初始化即可。在有了 DistMeta 之后，与 AutoAdvanced 的实现类似，只需两行代码就能将 DistAdvanced 实现完毕。

```
01 # 导入上一章实现的 AutoBase 与 AdvancedNN 的结合: AutoAdvanced
02 from Dist.NeuralNetworks.f AutoNN.DistNN import AutoAdvanced
03 # 导入前三节与刚刚实现的 DistMixin 与 DistMeta
04 from Dist.NeuralNetworks.DistBase import DistMixin, DistMeta
05
06 # 实现可以当作自动机器学习模型使用的 DistAdvanced
07 # 注意一定要把 AutoAdvanced 放在第一位，把 DistMixin 放在第二位
08 class DistAdvanced(AutoAdvanced, DistMixin, metaclass=DistMeta):
09     pass
```

实现完 DistAdvanced 之后，我们就可以正式开始进行实验了。由于要和 AutoSklearn 进行对比，所以公平起见，我们就使用论文中用来做实验的数据集来进行实验，这样能更客观地比较 DistAdvanced 与 AutoSklearn 的性能。而且幸运的是，由于 AutoSklearn 用来做实验的数据集都是 OpenML 网站上的数据集，而且 OpenML 提供了 Python 的 API，所以我们可以通过 Python 脚本来下载所有数据集并加以应用。下面我们依次介绍如何用脚本来下载数据集以及有了数据集之后如何训练 DistAdvanced 模型，然后我们会在最后给出 DistAdvanced 的实际表现以及这些表现和 AutoSklearn 的表现的对比。

注意：完整的、使用 DistAdvanced 在 OpenML 数据集上进行实验的代码可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/DistNN/TestOpenML.py。

OpenML 这个网站在 1.1.2 节中就已经提到过，它的主页是 <https://www.openml.org/>，里面有大量的人造数据集和真实数据集。它的每一个数据集都与一个 id 相对应，想要获取某个特定 id 的数据集时只需访问 <https://www.openml.org/d/xxx> 即可，其中 xxx 代指那个特定的 id。比如我想获取 sick 数据集，而 sick 数据集的 id 是 38，那么就可以通过 <https://www.openml.org/d/38> 来访问这个数据集。

如果只关心其中一个数据集的话，手动去访问相应的网址倒也不算麻烦；但是如果要在很多数据集上进行实验的话，就必须将数据集的下载过程自动化。由于 OpenML 已经提供了很多方便的 API，相应的代码实现只需按照其官方文档 (<https://openml.github.io/openml-python/stable/>) 来写就行，如代码 7.3 所示。

代码 7.3 OpenML 实验代码: _Tests/DistNN/TestOpenML.py

```
01 # OpenML 的安装方法可以参见上述官方文档对应的链接
02 from openml import datasets
03
04 # 将 AutoSklearn 论文中用到的数据集对应的 id 全部记录在一个 list 中
```

```
05 IDS = [  
06     38, 46, 179,  
07     184, 389, 554,  
08     772, 917, 1049,  
09     1111, 1120, 1128,  
10     293,  
11 ]  
12  
13 # 定义一个从 OpenML 网站上下载数据集及相应数据集信息的函数  
14 def download_data():  
15     # 将数据集的保存地点设置为“Data”  
16     data_folder = "Data"  
17     # 由于 OpenML 可以提供每个数据集对应的 numerical idx  
18     # 所以我们需要创建一个文件夹来存储它们  
19     idx_folder = os.path.join(data_folder, "idx")  
20     # 如果数据集文件夹和 numerical idx 文件夹不存在的话, 就创建它们  
21     if not os.path.isdir(data_folder):  
22         os.makedirs(data_folder)  
23     if not os.path.isdir(idx_folder):  
24         os.makedirs(idx_folder)  
25     # 遍历所有需要下载的数据集对应的 id  
26     for idx in IDS:  
27         # 输出一些信息以告知用户现在正在下载的数据集  
28         print("Downloading {}".format(idx))  
29         # 定义数据文件的名字  
30         data_file = os.path.join(data_folder, "{}.txt".format(idx))  
31         # 定义 numerical idx 文件的名字  
32         idx_file = os.path.join(idx_folder, "{}.npz".format(idx))  
33         # 如果这两个文件都已经存在的话  
34         # 说明我们已经下载过当前数据集  
35         if os.path.isfile(data_file) and os.path.isfile(idx_file):  
36             # 所以直接跳过即可  
37             continue  
38         # 否则, 就需要利用第 01 行 import 进来的 datasets 的相应方法  
39         # 从 OpenML 网站上把当前数据集下载下来  
40         dataset = datasets.get_dataset(idx)  
41         # 利用下载下来的数据集对象的 get_data 方法  
42         # 获取原始数据、它的 categorical_idx 以及各个维度的数据的名字  
43         data, categorical_idx, names = dataset.get_data(  
44             return_categorical_indicator=True,  
45             return_attribute_names=True  
46         )  
47         # 考虑到 OpenML 的原始数据有可能是用稀疏矩阵进行表示的  
48         # 所以当 data 不是 numpy 数组时, 我们要调用稀疏矩阵的相应方法  
49         # 来将 data 转为 numpy 数组  
50         data = data.toarray() if not isinstance(data, np.ndarray) else data  
51         # 利用各个维度的数据的名字和标签的名字  
52         # 来获取标签所在列  
53         target_idx = names.index(dataset.default_target_attribute)
```

```

54     # 利用 categorical idx 来获取 numerical idx
55     # 顾名思义, 直接用逻辑非 (“~”) 即可完成转换
56     numerical_idx = ~np.array(categorical_idx)
57     # 由于我们实现 AutoBase 时默认标签位于最后一列
58     # 所以需要调用 swap 方法来将标签换到最后一列
59     # swap 方法的说明马上在后文进行
60     swap(numerical_idx, target_idx, -1)
61     swap(data, target_idx, -1)
62     # 将原始数据集写入数据文件
63     with open(data_file, "w") as file:
64         file.write("\n".join([
65             " ".join(map(lambda n: str(n), line)) for line in data]))
66     # 保存 numerical idx
67     np.save(idx_file, numerical_idx)
68
69 # 定义一个将 numpy 数组的第 i1 列与第 i2 列互换的方法
70 def swap(arr, i1, i2):
71     arr[..., i1], arr[..., i2] = arr[..., i2], arr[..., i1].copy()

```

运行这段代码之后, 就会在原地创建一个名为 “_Data” 的文件夹, 里面放着各个 id 对应的数据集的数据文件, 以及这些数据集的 numerical_idx 对应的 numpy 数组文件, 如图 7.10~图 7.12 所示。

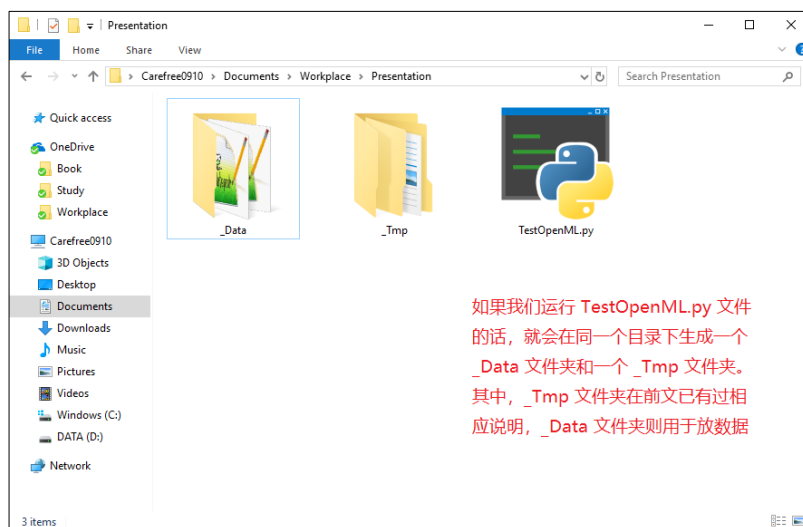


图 7.10 下载好的 OpenML 数据集所处的路径 (1)



图7.11 下载好的OpenML数据集所处的路径 (2)

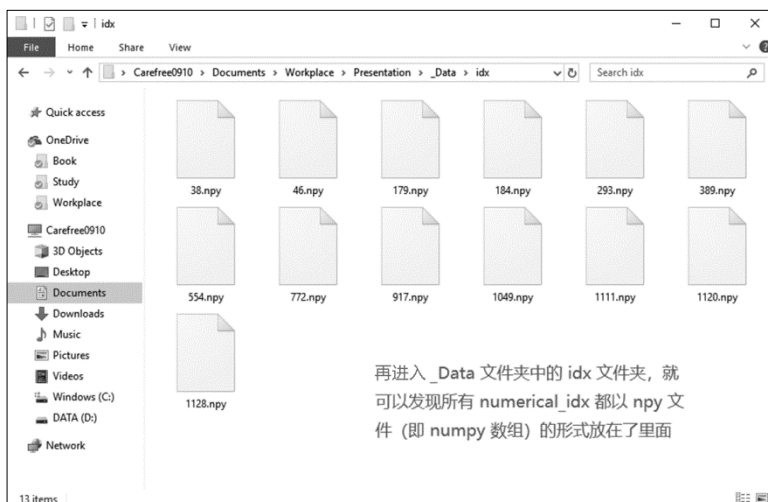


图7.12 下载好的OpenML数据集的numerical_idx所处的路径

有了这些文件的话, 使用 DistAdvanced 来进行实验本身就是非常容易的事情了, 代码如下所示。

```

72 # 导入需要用到的第三方库
73 import copy
74 import numpy as np
75 import tensorflow as tf
76
77 # 导入 DistAdvanced 模型
78 from _Dist.NeuralNetworks.g_DistNN.NN import DistAdvanced
79
80 # 定义利用 DistAdvanced 来进行实验的主函数
81 def main():
82     # 遍历所有需要进行实验的 id

```

```

83     for idx in IDS:
84         # 获取 numerical idx
85         numerical_idx = np.load("Data/idx/{}.npz".format(idx))
86         # 设置好名字参数与 numerical_idx 参数
87         local_params = {
88             "name": str(idx),
89             "data info": {"numerical_idx": numerical_idx}
90         }
91         # 先根据设置好的参数, 创建一个 DistAdvanced 模型
92         # 然后调用 empirical_search 方法进行参数搜索
93         # 最后调用 k_random 方法来对搜索到的参数进行评估
94         DistAdvanced(*local_params).empirical_search(
95             cv_rate=0.1, test_rate=0.1).k_random(
96             cv_rate=0.1, test_rate=0.1)

```

可以看到, 其实只需一行代码就能完成 DistAdvanced 模型的创建、参数的搜索以及参数搜索完之后的综合评估 (之所以上述代码中分成了 94~96 三行代码仅仅是为了排版好看), 这比起其他许多自动机器学习模型而言都是较为便捷的。

但是, 仅仅是调用上的便捷并不足以成为我们使用它的理由, 它还必须在实际表现上优于其余模型才行。而由于 AutoSklearn 是开源算法, 而且我们在参数搜索时会将所用的训练集 X_{train} 、交叉验证集 X_{cv} 与测试集 X_{te} 保存在临时数据文件夹中, 所以只需用 AutoSklearn 在这些相同的 X_{train} 和 X_{cv} 上进行训练, 并对比它在 X_{te} 上的表现与 DistAdvanced 在 X_{te} 上的表现即可。当统一把参数搜索时间限制在一个小时时, AutoSklearn 与 DistAdvanced 的表现对比如图 7.13 所示。

	Task	Ptr	N	Auto-Sklearn	DistAdvanced
38	Binary	3395	29	99.17	99.72
46	3-classes	2871	60	99.44	99.74
179	Binary	43958	14	88.08	91.78
184	18-classes	25250	6	98.51	99.50
293	Binary	522911	54	99.34	98.50
389	17-classes	2217	2000	98.89	98.32
554	10-classes	63000	784	99.89	99.95
772	Binary	1960	3	49.91	51.03
917	Binary	900	25	90.70	75.44
1049	Binary	1312	37	89.93	95.02
1111	Binary	45000	230	71.68	79.14
1120	Binary	17118	10	93.02	93.32
1128	Binary	1390	10935	97.92	99.86

图 7.13 AutoSklearn 与 DistAdvanced 在 OpenML 数据集上的对比 (评估指标: auc)

可以看到在大多数情况下, 我们的 DistAdvanced 模型都会比 AutoSklearn 开源出来的模型要好, 这说明第 5 章介绍的诸多技术、第 6 章介绍的半自动化机器学习框架以及本章介绍的工程化机器学习框架确实有着较为强大的性能。

注意: 值得一提的是, 虽然我们在整整 13 个数据集上进行了实验, 而且每个数据集上都只使用了级别 3 的参数搜索 (即只搜索了 16 个参数组合), 但是这 13 个数据集上的最优参数组合基本可以说是各不相同的。这恰恰就佐证了第 6 章所说的“没有免费的午餐定理”, 以及说明了本章介绍的参数搜索算法的重要性。

至此, 本书的主要内容就告一段落了。大体上来说, 前 5 章偏重理论、后两章偏重实践。

其中，前 5 章写起来和上一本书写起来的感觉大抵还相去不远，但后两章写起来的感觉则颇为陌生。虽然说前 5 章是这本书的根基，但笔者个人认为后两章的内容反而会是实际应用中经常会用到的知识，因为它们已经基本脱离了算法而迈向了工程。不过，毕竟后两章在写的时候总感觉捉襟见肘，所以实际的效果可能会不尽如人意；如果大家有什么意见、建议或疑问的话，都欢迎随时反馈到我的邮箱：syameimaru.saki@gmail.com，我会尽快做出回应。

7.5 本章小结

- 使用 logging 能够较好地管理输出信息。
- 由于多次实验会涉及很多功能的交叉，所以一个好的编程范式就是创建临时数据文件夹来开展相应的工作。
- 前沿的参数搜索方法有贝叶斯优化与强化学习，但它们都不太适用于（使用神经网络来解决）结构化数据的分类任务。再加上我们拥有 WnD、DNDF 与剪枝等技术，所以直接使用朴素的网格搜索或随机搜索反而会是更好的选择。

附录 A

SVM 的 TensorFlow 实现

在本书的第 3 章，我们实现了 TensorFlow 模型的基本框架 Base，并指出了只要是能够基于梯度下降法训练的模型，都能用 Base 来进行相应的实现。而由第 2 章的相应讨论可知，无论是原始形式的支持向量机（LinearSVM），还是应用了核方法之后的支持向量机（SVM），它们都能够用梯度下降法来进行求解，所以利用 Base 理应能够实现出这两个模型。本附录的主要目的，就是证明 Base 确实能够用来实现 LinearSVM 与 SVM。而除此之外，我们还在第 6 章、第 7 章中指出过，实现的半自动化机器学习框架 AutoBase 与工程化机器学习框架 DistMixin 都能方便地应用在基于 Base 实现的 model 之上，所以本附录还会通过实现“半自动化支持向量机（AutoLinearSVM）”和“工程化支持向量机（DistLinearSVM）”来证明这一点。

本附录将涵盖如下内容：

- LinearSVM 和 SVM 的（基于 Base 的）TensorFlow 实现
- AutoLinearSVM 和 DistLinearSVM 的实现
- DistLinearSVM 的实际性能

A.1 利用 Base 实现 LinearSVM

与神经网络不同，LinearSVM 的超参数相对而言非常少，只有对放松的“惩罚力度” C 这个超参数。正因此，在有了 Base 的前提下，LinearSVM 的初始化以及各个超参数的初始化方法将会非常简单，如代码 A.1 所示。

代码 A.1 LinearSVM 的 TensorFlow 实现：b_TraditionalML/SVM.py

```
01 # 继承第 3 章实现的 TensorFlow 模型的基本框架 Base
02 class LinearSVM(Base):
03     def __init__(self, *args, **kwargs):
```

```

04     # 继承 Base 的初始化方法, 完成绝大部分的初始化工作
05     super(LinearSVM, self).init(*args, **kwargs)
06     # 将名字改成 LinearSVM
07     self.name.appendix = "LinearSVM"
08     # 定义一个记录惩罚力度 C 的属性
09     self.c = None
10
11     # 定义利用数据来初始化超参数的方法
12     def init from data(self, x, y, x test, y test, sample weights, names):
13         # 仍然是继承 Base 的相应方法来完成大部分工作
14         super(LinearSVM, self).init from data(
15             x, y, x test, y test, sample weights, names)
16         # 由于支持向量机原始形式只能解决二分类问题
17         # 所以这里要特地将模型评估指标默认设置为 binary acc
18         metric = self.model param settings.setdefault("metric", "binary acc")
19         # 且如果用户指定了使用 acc 的话
20         if metric == "acc":
21             # 还要把它改成 binary acc
22             self.model param settings["metric"] = "binary acc"
23         # 同时将 n class 属性设置为 1, 这是因为支持向量机只输出一个数
24         # 从而输出层的神经元个数应该是 1
25         self.n class = 1
26
27     # 定义初始化模型超参数的方法
28     # 由于 LinearSVM 与神经网络有较大不同
29     # 所以我们需要更改一些默认参数设置
30     def init model param settings(self):
31         # 将学习速率默认设置为 0.01
32         self.model param settings.setdefault("lr", 0.01)
33         # 默认迭代 1000 次
34         self.model param settings.setdefault("n epoch", 10 ** 3)
35         # 默认最多迭代 100 万次
36         self.model param settings.setdefault("max epoch", 10 ** 6)
37         # 继承 Base 的相应方法完成剩余的参数初始化
38         super(LinearSVM, self).init_model_param_settings()
39         # 将惩罚力度默认设置为 1
40         self.c = self.model_param_settings.get("C", 1.)

```

在初始化完之后, 接着要做的就是模型的搭建和代价函数

$$C_{SVM}(w_0, \mathbf{w}, D) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^N |1 - y_i \cdot (w_0 + \mathbf{w}^T \mathbf{x}_i)|_+$$

的定义了, 它们的实现如下所示。

```

41     # 搭建 LinearSVM 模型
42     # 前 4 行与朴素神经网络处的实现一致, 故不赘述
43     def build model(self, net=None):
44         self.model built = True
45         if net is None:

```



```

46         net = self. tfx
47         current dimension = net.shape[1].value
48         # 由于 LinearSVM 本质是一个线性模型
49         # 所以直接用相应方法获取模型输入 (net) 的线性映射
50         # 来作为模型的输出 (self. output) 即可
51         self. output = self. fully connected linear(
52             net, [current dimension, 1], " final projection")
53
54     # 定义 LinearSVM 的代价函数
55     def define loss and train step(self):
56         self. loss = self.c * tf.reduce sum(
57             tf.maximum(0., 1 - self. tfy * self. output)
58         ) + tf.nn.l2 loss(self. ws[0])
59         # 调用损失函数与优化器来获取 “训练步骤” self. train step
60         with tf.control dependencies(tf.get collection(
61             tf.GraphKeys.UPDATE OPS)):
62             self._train_step = self. optimizer.minimize(self._loss)

```

仅就 LinearSVM 模型本身而言，至此其实就已经完成了所有的实现。不过，考虑到支持向量机要求数据集中的标签为 1 或 -1，所以我们需要把之前所做的假设——二分类时数据集的标签为 0 或 1 这一点做出修正。具体而言，我们只需在用到标签的地方将所有的 0 都改成 -1 即可：

```

63     def get feed dict(self, x, y=None, weights=None, is training=False):
64         # 如果提供了标签的话
65         if y is not None:
66             # 就把标签中的 0 全部改成 -1
67             y[y == 0] = -1
68         return super(LinearSVM, self)._get_feed_dict(x, y, weights, is_training)

```

同时，由于模型在接收一个输入后只会输出一个数（而不像神经网络那样输出一个二维向量），且该数的正负符号决定了模型认为输入应该对应的类别，所以我们需要重新定义一下模型预测数据类别的方法：

```

69     def predict classes(self, x):
70         # 根据输出是否大于 0 来决定输入属于哪一类
71         return (self. calculate(
72             x, tensor=self._output, is_training=False) >= 0).astype(np.int32)

```

以上就是 LinearSVM 的所有实现，可以看到需要做出改动的地方很少，整体也确实清晰可读。

注意：LinearSVM 在实际问题上的性能会在 A.3 节中（结合第 7 章用过的诸多 OpenML 数据集）给出。

A.2 利用 Base 实现 SVM

SVM 与 LinearSVM 在算法上大同小异，所以在实现的思路上也大致相同。只不过，在处理（应用了核方法的）SVM 时需要额外注意如下两个地方：

- 高效地实现核函数以免在此花费过多的时间。
- 维护训练集的核矩阵，从而加速训练与预测的过程。

注意：本节所实现的 SVM 只能处理问题规模较小的情形，因为当问题规模变大之后一般还是用神经网络比较好。如果实在想要用 SVM 处理大规模问题的话，也可以调用相应的机器学习第三方库，不过那就不在本书的讨论范围以内了。

所以比起 LinearSVM 的实现来说，SVM 的实现处处都有核方法与核矩阵的身影。首先来看看初始化的相应方法。

```

01 # 由于 SVM 与 LinearSVM 有很多相通之处
02 # 所以继承 LinearSVM 会比继承 Base 要更方便
03 class SVM(LinearSVM):
04     def __init__(self, *args, **kwargs):
05         super(SVM, self).__init__(*args, **kwargs)
06         # 将名字改成 SVM
07         self._name_appendix = "SVM"
08         # 定义核函数相关的超参数对应的属性
09         self._p = self._gamma = None
10         # 定义核矩阵相关的属性
11         self._x = self._gram = self._kernel_name = None
12
13     def init_from_data(self, x, y, x_test, y_test, sample_weights, names):
14         # 记录下特征矩阵
15         self._x = np.atleast_2d(x).astype(np.float32)
16         y = np.asarray(y, np.float32)
17         # 获取多项式核的参数 p
18         self._p = self.model_param_settings.setdefault("p", 3)
19         # 获取径向基核的参数  $\gamma$ 
20         self._gamma = self.model_param_settings.setdefault(
21             "gamma", 1 / self._x.shape[1])
22         # 获取所选的核函数，默认为径向基核
23         self._kernel_name = self.model_param_settings.setdefault(
24             "kernel name", "rbf")
25         # 根据核函数对应的 property (self.kernel)，计算出核矩阵
26         # 该 property 的相应说明会放在后文进行
27         self._gram = self.kernel(self._x, self._x)
28         x_test = self.kernel(x_test, self._x)
29         super(SVM, self).init_from_data(
30             self._gram, y, x_test, y_test, sample_weights, names)
31
32     def init_model_param_settings(self):
33         super(SVM, self).init_model_param_settings()
34         # 设置相应的属性
35         self._p = self.model_param_settings["p"]
36         self._gamma = self.model_param_settings["gamma"]
37         self._kernel_name = self.model_param_settings["kernel name"]

```

注意在上述代码的第 27 行到第 30 行处，我们把训练集、测试集对应的核矩阵当作了 Base

的训练集、测试集来进行输入,这是因为核方法的本质可以被看作一种特殊的数据预处理过程。也正因此,我们在进行评估或预测时,也要先把原始特征矩阵转换为核矩阵之后再调用先前实现过的评估或预测方法。

```

38     def evaluate(self,
39         x=None, y=None, x_cv=None, y_cv=None,
40         x_test=None, y_test=None, metric=None
41     ):
42         # 将需要进行转换的特征矩阵转换为核矩阵
43         n_sample = self.x.shape[0]
44         cv_feat_dim = None if x_cv is None else x_cv.shape[1]
45         test_feat_dim = None if x_test is None else x_test.shape[1]
46         if x_cv is None:
47             x_cv = None
48         elif cv_feat_dim != n_sample:
49             x_cv = self.kernel(x_cv, self.x)
50         if x_test is None:
51             x_test = None
52         elif test_feat_dim != n_sample:
53             x_test = self.kernel(x_test, self.x)
54         # 将转换后的特征输入先前实现的方法,完成模型的评估
55         return super(SVM, self).evaluate(x, y, x_cv, y_cv, x_test, y_test)
56
57     def predict(self, x):
58         # 先转换为核矩阵,再完成预测
59         return self.predict(self.kernel(x, self.x))
60
61     def predict_classes(self, x):
62         # 这里的实现和 LinearSVM 处的实现思想一致
63         return (self.predict(x) >= 0).astype(np.int32)
64
65     def evaluate(self,
66         x, y, x_cv=None, y_cv=None,
67         x_test=None, y_test=None, metric=None
68     ):
69         # 先转换为核矩阵,再完成评估
70         return self.evaluate(
71             self.kernel(x, self.x), y, x_cv, y_cv, x_test, y_test, metric)

```

在完成了初始化之后,与 LinearSVM 类似,接下来就要搭建模型以及定义代价函数了。不过正如前文所说,由于核方法的应用可以看作特殊的数据预处理,所以 SVM 的模型搭建与 LinearSVM 的模型搭建是一致的。而我们实现 SVM 时又继承了 LinearSVM,所以就无须显式地实现模型的搭建过程了。至于代价函数的定义,就只需按照

$$C_{SVM}(G, D) = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) + C \sum_{i=1}^N \left| 1 - \mathbf{y}_i \cdot \left(\mathbf{w}_0 + \sum_{j=1}^N \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) \right) \right|_+$$

来进行相应的实现即可:

```

72     def define_loss_and_train_step(self):
73         self.loss = self.c * tf.reduce_sum(
74             tf.maximum(0., 1 - self.tfy * self.output)
75         ) + 0.5 * tf.matmul(
76             self.ws[0], tf.matmul(self.gram, self.ws[0]),
77             transpose_a=True
78         )[0]
79         with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
80             self._train_step = self._optimizer.minimize(self._loss)

```

注意：由于我们将核方法视为了特殊的数据预处理，所以模型接收的特征矩阵将总会是核矩阵，从而权值矩阵（`self_ws[0]`）对应的将会是公式中的 α 而非 w 。

至此，SVM 本身的实现就完成了，下面我们来展示一下各个核函数的高效的实现方式，这里面用到了一些 `numpy` 的特殊技巧，所以理解的难度可能会比较大。

```

81     # 定义线性核
82     @staticmethod
83     def linear(x, y):
84         return x.dot(y.T)
85
86     # 定义多项式核
87     @staticmethod
88     def poly(x, y, p):
89         return (x.dot(y.T) + 1) ** p
90
91     # 定义径向基核
92     @staticmethod
93     def rbf(x, y, gamma):
94         return np.exp(-gamma * np.sum((x[..., None, :] - y) ** 2, axis=2))
95
96     # 定义调用核函数的 property
97     @property
98     def kernel(self):
99         if self.kernel_name == "linear":
100             return self.linear
101         if self.kernel_name == "poly":
102             return lambda x, y: self.poly(x, y, self.p)
103         if self.kernel_name == "rbf":
104             return lambda x, y: self.rbf(x, y, self._gamma)

```

以上就是 SVM 的所有实现。由于它引入了核方法这样一个特殊的数据预处理过程，所以相应的实现要显得复杂一些，但大体上而言还是比较容易理解的。

A.3 LinearSVM 的实际性能

由第 6 章和第 7 章中 `AutoAdvanced` 与 `DistAdvanced` 的实现可知，在有了基于 `Base` 实现的

model 之后，应用 AutoBase 和 DistMixin 都是非常简单的：只需依次继承相应的类并把元类设置好即可。在 LinearSVM 里，我们可以看到该说法确实成立：

```
01 # 实现“半自动化 LinearSVM”
02 class AutoLinearSVM(AutoBase, LinearSVM, metaclass=AutoMeta):
03     pass
04
05 # 实现“工程化 LinearSVM”
06 class DistLinearSVM(AutoLinearSVM, DistMixin, metaclass=DistMeta):
07     pass
```

在有了 DistLinearSVM 之后，我们就能用第 7 章中用到过的诸多 OpenML 数据集中的二分类数据集来验证基于 TensorFlow 实现的 LinearSVM 的性能了。具体的实验代码与第 7 章所用的实验代码除了建模的步骤不同外（将 DistAdvanced 换成 DistLinearSVM，同时直接进行多次实验而不用进行参数搜索），其余部分完全一致。对源代码感兴趣的读者可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/_Tests/SVM/TestOpenML.py，这里我们就直接给出结果，如图 A.1 所示。

	Task	Ptr	N	LinearSVM	DistAdvanced
38	Binary	3395	29	96.50	97.82
179	Binary	43958	14	84.16	86.25
293	Binary	522911	54	76.27	93.52
772	Binary	1960	3	58.56	58.00
917	Binary	900	25	59.22	62.67
1049	Binary	1312	37	91.78	89.50
1111	Binary	45000	230	98.02	98.02
1120	Binary	17118	10	78.80	87.54
1128	Binary	1390	10935	94.91	97.78

图 A.1 LinearSVM 与 DistAdvanced 在 OpenML 二分类数据集上的对比（评估指标：acc）

可以看到，虽然大都不如我们实现的可以看作自动机器学习模型的 DistAdvanced，但是作为一个本质为线性模型的 LinearSVM 来说，其表现还是相当不错的。

附录 B

numba 的基本应用

Python 虽然是很强大的语言，但是它的运行速度一直是其痛点。虽然我们都知道 Python 比较慢，但很多时候不知道为什么这么慢。在笔者个人看来，Python 之所以这么慢恰恰就是因为它最大的优势之处——它太动态了。如果我们能够事先编译一下 Python，让它静态一点，速度理应就会上来很多。

这其实就是著名的标准库——Cython 的目标。然而 Cython 毕竟不是原生的 Python 代码，使用起来不仅对于纯 Python 代码工作者很不友好，对熟知 C++ 等底层语言的程序员而言也仍有诸多不便。为此，numba 就成了一个功能强大又容易上手的替代选择。

本附录将涵盖如下内容：

- 如何加速 Python 中低效的循环语句
- 如何实现 numpy 中的广播功能
- 如何实现高效的并发（多线程）

注意：本附录涉及的代码的运行环境均为 Jupyter Notebook、Python 3.6.1。

B.1 使用 jit 加速低效的 for 语句

jit 的全称是 Just-in-time，在 numba 里面则特指 Just-in-time compilation（即时编译）。它背后的原理我们就不展开叙述了，但从直观上来说的话，从其名字就不难看出它是我们想要的东西。

下面来看一个简单的例子——给数组中的每个数加上一个常数 c。

```
01 import numba as nb
02 import numpy as np
```

```

03
04 # 普通的 for
05 def add1(x, c):
06     rs = [0.] * len(x)
07     for i, xx in enumerate(x):
08         rs[i] = xx + c
09     return rs
10
11 # 列表表达式 (list comprehension)
12 def add2(x, c):
13     return [xx + c for xx in x]
14
15 # 使用 jit 加速后的 for
16 @nb.jit(nopython=True)
17 def add_with_jit(x, c):
18     rs = [0.] * len(x)
19     for i, xx in enumerate(x):
20         rs[i] = xx + c
21     return rs
22
23 # 生成测试用数据
24 y = np.random.random(10**5).astype(np.float32)
25 x = y.tolist()
26
27 # 计时
28 assert np.allclose(add1(x, 1), add2(x, 1), add with jit(x, 1))
29 %timeit add1(x, 1)
30 %timeit add2(x, 1)
31 %timeit add with jit(x, 1)
32 print(np.allclose(wrong_add(x, 1), 1))

```

以下是程序运行结果:

```

9.92 ms ± 188 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
5.77 ms ± 347 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
3.48 ms ± 171 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```

可以看到, 使用 numba 将会有显著的性能提升。这里需要注意的是:

- jit 能够加速的不限于 for, 但一般而言加速 for 会比较常见, 效果也比较显著。笔者在个人实现的 numpy 版本的卷积神经网络(CNN)中用了 jit 后, 可以直接把代码加速 20 倍左右。具体代码可以参见 <https://github.com/carefree0910/MachineLearning/blob/master/NN/Basic/Layers.py#L9>, 不过如果不想看源代码的话, 也可以参见 <https://github.com/carefree0910/MachineLearning/blob/master/Notebooks/numba/zh-cn/CNN.ipynb> 这个 jupyter notebook, 笔者在其中做了一些比较简单的实验。
- jit 会在某种程度上“预编译”你的代码, 这意味着它会在某种程度上固定住各个变量的数据类型; 所以在 jit 下定义数组时, 如果想要使用的是 float 数组的话, 就不能用[0]

* len(x) 的定义，应该像上面那样在 0 后面加一个小数点：[0.] * len(x)。

B.2 使用 vectorize 实现 Ufunc 功能

虽然 jit 确实能让代码加速不少，但比 numpy 的 Ufunc（广播）还是要差很多：

```
01 assert np.allclose(y + 1, add with jit(x, 1))
02 %timeit add with jit(x, 1)
03 %timeit y + 1
```

结果将会是：

```
3.76 ms ± 292 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
19.8 µs ± 426 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

可以看到几乎有 200 倍的差距，这当然是无法忍受的。为此，我们可以用 vectorize 来定义出类似 Ufunc 的函数：

```
01 @nb.vectorize(nopython=True)
02 def add with vec(yy, c):
03     return yy + c
04
05 assert np.allclose(y + 1, add with vec(y, 1), add with vec(y, 1.))
06 %timeit add with vec(y, 1)
07 %timeit add with vec(y, 1.)
08 %timeit y + 1
09 %timeit y + 1.
```

上述程序的运行结果将会是：

```
72.5 µs ± 3.46 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
64.2 µs ± 1.9 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
24.6 µs ± 1.81 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
25.3 µs ± 1.61 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

虽然还是有 2 倍左右的差距，但已经好很多了。这里仍然有几个需要注意的地方：

- vectorize “装饰”的函数所接受的参数是一个个的数而非整个数组，所以上述 add_with_vec 的参数 yy 其实是输入数组 y 中的元素而不是 y 本身。更详细的说明可以参见官方文档：<http://numba.pydata.org/numba-doc/0.23.0/user/vectorize.html>。
- 可以看到当常数 c 是整数或浮点数时，速度是不同的。笔者个人猜测这是因为若常数 c 为整数，那么实际运算时需要将它转化为浮点数，从而导致速度变慢。

此外，上述代码中我们没有显式地定义函数的参数类型和返回类型，但可以预先定义。比如，如果确定常数 c 是整数的话，就可以这样写：

```
01 @nb.vectorize("float32(float32, int32)", nopython=True)
02 def add with vec(yy, c):
01 return yy + c
```


而如果确定常数 `c` 是浮点数的话，就可以这样写：

```
01 @nb.vectorize("float32(float32, float32)", nopython=True)
02 def add with vec(yy, c):
02     return yy + c
```

而如果确定常数 `c` 不是整数就是浮点数的话，就可以这样写：

```
01 @nb.vectorize([
02     "float32(float32, int32)",
03     "float32(float32, float32)"
04 ], nopython=True)
05 def add with vec(yy, c):
06     return yy + c
```

注意：`float32` 和 `float64`、`int32` 和 `int64` 是不同的，需要小心。

此外，`vectorize` 还有一个非常强大的功能，那就是它可以“并行（parallel）”：

```
01 @nb.vectorize("float32(float32, float32)", target="parallel", nopython=True)
02 def add with vec(y, c):
03     return y + c
04
05 assert np.allclose(y+1, add with vec(y,1.))
06 %timeit add with vec(y, 1.)
07 %timeit y + 1
```

虽说在普通的 Python 3.6.1 下，这段代码的运行结果将如下所示：

```
73.5 µs ± 4.22 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
21.2 µs ± 734 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

速度似乎甚至还变慢了。不过如果使用 Intel Distribution for Python 的话，会发现 `parallel` 版本甚至会比 `numpy` 原生的版本要稍快一些。

那么是否有用 `parallel` 总会更好的例子呢？当然是有的。

```
01 # 将数组中的所有元素限制在某个区间[a, b]内
02 # 小于 a 则置为 a，大于 b 则置为 b
03 # 经典应用：ReLU
04
05 @nb.vectorize(
06     "float32(float32, float32, float32)", target="parallel", nopython=True)
07 def clip with parallel(y, a, b):
08     if y < a:
09         return a
10     if y > b:
11         return b
12     return y
13
14 @nb.vectorize("float32(float32, float32, float32)", nopython=True)
15 def clip(y, a, b):
```

```

16     if y < a:
17         return a
18     if y > b:
19         return b
20     return y
21
22 assert np.allclose(
23     np.clip(y, 0.1, 0.9), clip(y, 0.1, 0.9), clip with parallel(y, 0.1, 0.9))
24 %timeit clip with parallel(y, 0.1, 0.9)
25 %timeit clip(y, 0.1, 0.9)
26 %timeit np.clip(y, 0.1, 0.9)

```

上述程序的运行结果将会是：

```

95.2 µs ± 5.6 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
104 µs ± 4.52 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
377 µs ± 62 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

```

这个例子中的性能提升就是实打实的了。总之，使用 `parallel` 时不能一概而论，还是要做些实验来看一下实际情况。

需要指出的是，`vectorize` 中的参数 `target` 一共有三种取值：`cpu`（默认）、`parallel` 和 `cuda`。关于选择哪个值，官方文档上有很好的说明：“A general guideline is to choose different targets for different data sizes and algorithms. The “cpu” target works well for small data sizes (approx. less than 1KB) and low compute intensity algorithms. It has the least amount of overhead. The “parallel” target works well for medium data sizes (approx. less than 1MB). Threading adds a small delay. The “cuda” target works well for big data sizes (approx. greater than 1MB) and high compute intensity algorithms. Transferring memory to and from the GPU adds significant overhead.”

B.3 利用 `jit(nogil=True)` 实现高效并发

我们知道，Python 中由于有 GIL 的存在，所以多线程用起来非常不舒服。不过 `numba` 的 `jit` 里面有一个参数叫 `nogil`，顾名思义，它能在一定程度上缓解 GIL 带来的性能损伤。

下面就来看一个例子。

```

01 import math
02 from concurrent.futures import ThreadPoolExecutor
03
04 # 计算类似 Sigmoid 函数
05 def np_func(a, b):
06     return 1 / (a + np.exp(-b))
07
08 # 参数中的 result 代表的即是我们想要的结果，后同
09 # 第一个 kernel, nogil 参数被设为了 False
10 @nb.jit(nopython=True, nogil=False)
11 def kernell(result, a, b):

```

```

12     for i in range(len(result)):
13         result[i] = 1 / (a[i] + math.exp(-b[i]))
14
15 # 第二个 kernel, nogil 参数被设为了 True
16 @nb.jit(nopython=True, nogil=True)
17 def kernel2(result, a, b):
18     for i in range(len(result)):
19         result[i] = 1 / (a[i] + math.exp(-b[i]))
20
21 def make_single_task(kernel):
22     def func(length, *args):
23         result = np.empty(length, dtype=np.float32)
24         kernel(result, *args)
25         return result
26     return func
27
28 def make_multi_task(kernel, n_thread):
29     def func(length, *args):
30         result = np.empty(length, dtype=np.float32)
31         args = (result,) + args
32         # 将每个线程接受的参数定义好
33         chunk_size = (length + n_thread - 1) // n_thread
34         chunks = [
35             [arg[i*chunk_size:(i+1)*chunk_size]
36              for i in range(n_thread)] for arg in args
37         ]
38         # 利用 ThreadPoolExecutor 进行并发
39         with ThreadPoolExecutor(max_workers=n_thread) as e:
40             for _ in e.map(kernel, *chunks):
41                 pass
42         return result
43     return func
44
45 length = 10 ** 6
46 a = np.random.rand(length).astype(np.float32)
47 b = np.random.rand(length).astype(np.float32)
48
49 nb_func1 = make_single_task(kernel1)
50 nb_func2 = make_multi_task(kernel1, 4)
51 nb_func3 = make_single_task(kernel2)
52 nb_func4 = make_multi_task(kernel2, 4)
53
54 rs_np = np_func(a, b)
55 rs_nb1 = nb_func1(length, a, b)
56 rs_nb2 = nb_func2(length, a, b)
57 rs_nb3 = nb_func3(length, a, b)
58 rs_nb4 = nb_func4(length, a, b)
59 assert np.allclose(rs_np, rs_nb1, rs_nb2, rs_nb3, rs_nb4)

```

```
60 %timeit np_func(a, b)
61 %timeit nb_func1(length, a, b)
62 %timeit nb_func2(length, a, b)
63 %timeit nb_func3(length, a, b)
64 %timeit nb_func4(length, a, b)
```

这个例子相当长，不过我们只需要知道如下两点即可：

- `make_single_task` 和 `make_multi_task` 分别生成单线程函数和多线程函数。
- 生成的函数会调用相应的 `kernel` 来完成计算。

上述程序的运行结果将会是：

```
14.9 ms ± 538 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
8.32 ms ± 259 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
10.2 ms ± 368 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
8.25 ms ± 279 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
4.68 ms ± 114 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

一般来说，数据量越大，并发的效果越明显。反之，数据量小的时候，并发反而很有可能会降低性能。

附录 C

装饰器的基本应用

万物皆对象——Python。

正如第 6 章介绍元类时所说过的，Python 中的一大思想就是“万物皆对象”。其中，第 6 章介绍的元类告诉了我们“类亦对象”，而本附录所将介绍的装饰器，则会告诉大家“函数亦对象”。所谓的“函数亦对象”，意味着函数可以被赋值给变量，通过变量也能调用该函数。举一个例子：

```
01 def func(x):
02     return x + 1
03
04 plus_one = func
05 print(plus_one(1)) # 将会输出 2
```

装饰器的核心思想就是装饰函数这个对象，让函数在自身代码不变的情况下，增添一些具有普适性的功能。典型且我们打算进行讲解的一个例子就是，计算一个函数的运行时间。我们希望做到对任意一个函数，只要调用我们的装饰器，就能记录它的耗时，从而就可以进一步做性能分析与优化。

为此，我们先来看装饰器的基本用法：

```
01 @decorator
02 def func(*args, **kwargs):
03     .....
```

它等价于：

```
01 def func(*args, **kwargs):
02     .....
03 func = decorator(func)
```

其中，decorator 以函数对象为输入参数并返回一个函数对象。一般来说，其定义形如：

```
01 def decorator(func):
02     def wrapper(*args, **kwargs):
03         # 这里可以在调用函数之前做一些事
04         ans = func(*args, **kwargs)
05         # 这里可以在调用函数之后做一些事
06         return ans
07     return wrapper
```

注意：这里用到了许多 Python 的知识。关于可变参数*args 和**kwargs 的说明可以参见 <https://stackoverflow.com/questions/36901/what-does-double-star-asterisk-and-star-asterisk-do-for-parameters>，关于闭包（Closure）的说明则可以参见 <https://www.programiz.com/python-programming/closure>。

从而如果要对函数 func 的计时的话，只需再应用标准库 time 即可：

```
01 import time
02
03 def decorator(func):
04     def wrapper(*args, **kwargs):
05         # 记录下当前时间
06         t = time.time()
07         # 调用函数
08         ans = func(*args, **kwargs)
09         # 计算当前时间和先前时间的差，得出函数调用的耗时
10         t = time.time() - t
11         return ans, t
12     return wrapper
```

下面我们测试一下这个装饰器：

```
01 @decorator
02 def func():
03     for i in range(10 ** 6):
04         x = 0
05     return "Done"
06
07 func() # 将会输出类似于 ('Done', 0.02807450294494629) 的信息
```

当然，这个简单的版本其实还有许多缺点，包括但不限于：

- 装饰器本身无法传入参数。
- 函数的__name__属性发生了改变，导致执行有些依赖函数签名的代码时会出错。

不过 Python 的强大之处就是，当你实现一项功能觉得很棘手时，往往已经有现成的第三方库帮你解决了问题。在装饰器这里，wrapt 就是这样一个库，可以通过：

```
pip install wrapt
```

来安装它。其官方教程（<https://wrapt.readthedocs.io/en/latest/quick-start.html>）相当详尽，这里就

只说说它的基本用法。比如，上面我们定义的计时 `decorator` 在使用 `wrapt` 时会形如：

```
01 import time
02 import wrapt
03
04 @wrapt.decorator
05 def decorator(func, instance, args, kwargs):
06     t = time.time()
07     ans = func(*args, **kwargs)
08     t = time.time() - t
09     return ans, t
```

可以看到它少了一层嵌套，这就使得代码漂亮不少，同时几乎解决了所有装饰器可能带来的隐含问题，所以使用 `wrapt` 来定义装饰器可能总会是一个更好的选择。

下面我们来说说怎么给装饰器传参数。正如普通装饰器是返回函数对象的函数，要想给装饰器传参的话，就需要定义一个返回装饰器的函数。仍然以 `decorator` 为例，这次我们打算用一个阈值来判断函数 `func` 的运行快慢。

```
01 import time
02 import wrapt
03
04 def decorator(eps):
05     @wrapt.decorator
06     def wrapper(func, instance, args, kwargs):
07         t = time.time()
08         ans = func(*args, **kwargs)
09         t = time.time() - t
10         if t > eps:
11             print("Slow!")
12         else:
13             print("Fast!")
14         return ans, t
15     return wrapper
```

下面就来测试一下新的 `decorator`：

```
01 @decorator(0.01)
02 def func1():
03     for i in range(10 ** 6):
04         x = 0
05     return "Done"
06
07 @decorator(0.05)
08 def func2():
09     for i in range(10 ** 6):
10         x = 0
11     return "Done"
12
```

```
13 func1() # 将会输出“Slow!”以及之前将会输出的信息
14 func2() # 将会输出“Fast!”以及之前将会输出的信息
```

装饰器还有许多值得挖掘和探讨的地方，不过我们打算到此为止，因为以上的功能通常来说已经足够。在本附录的最后，我们来看一下它在笔者实现的神经网络里面的具体效果（如图 C.1 所示，代码则可以参见 <https://github.com/carefree0910/MachineLearning/blob/master/Util/Timing.py>）。

```
=====
Timing log
-----
Adam          [API]          run :      1.119907 s (Call Time: 57600)
ReLU          [Core]          bp :      0.6167095 s (Call Time: 19200)
ReLU          [Core]          activate : 1.004355 s (Call Time: 19266)
Softmax       [Core]          bp :      0.01103067 s (Call Time: 9600)
Softmax       [Core]          activate : 0.5301452 s (Call Time: 9633)
Log Likelihood [Core]          activate : 0.04110765 s (Call Time: 9633)
Neural Network [API]          fit :      5.941812 s (Call Time: 1)
Neural Network [Private Method] _opt :      2.122121 s (Call Time: 28800)
Neural Network [Private Method] _get_prediction : 0.02707005 s (Call Time: 33)
Neural Network [Private Method] _get_activations : 1.873269 s (Call Time: 9633)
=====
```

图C.1 利用装饰器实现的Timing的效果

附录 D

可视化

有的时候，即使我们对模型算法了然于胸，但具体到某个特定问题时为什么模型的表现很好（或很差）时，在不对模型与具体问题做进一步分析的情况下也不得而知。同时，理论上的东西虽然足够严谨，但在许多时候都是直观的东西更具有说服力。因此，本附录会介绍“可视化”这项技术，它既能让我们对数据集拥有更好的认识，也能让我们更直观地去理解模型的行为。

本附录将涵盖如下内容：

- 朴素贝叶斯的可视化
- 决策树的可视化

注意：支持向量机本质上是线性模型，即其本质是退化后的神经网络，而神经网络的可视化效果已在 4.3.2 节中的图 4.19 和图 4.20 中给出过，所以本附录就只叙述之前没有叙述过的朴素贝叶斯与决策树的可视化。

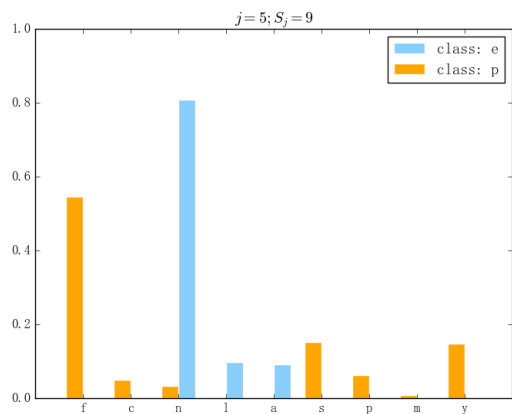
D.1 Naïve Bayes 的可视化

朴素贝叶斯是一个较为经典的贝叶斯分类器，它结合了许多统计学上的常用方法（极大似然估计、贝叶斯公式等）。虽然这在学过统计学的人看来是非常普通而好理解的，但是对于一般的人来说可能就不会这么认为了。事实上，笔者在开始接触朴素贝叶斯时还没有学过统计学的相关知识，当时那一头雾水的感觉至今难以忘记。

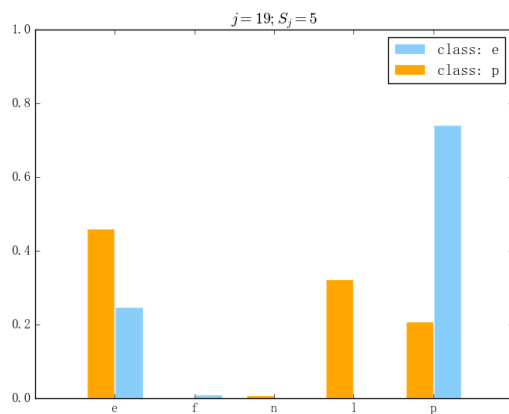
因此，如果能以一种更为直观的方式来说明朴素贝叶斯的工作原理的话，对于很多人来说可能都会是较大的助力。为此，我们可以对朴素贝叶斯进行一定的可视化。由第 2 章的相应讨论可知，朴素贝叶斯的原理用一句话来直观概括的话，就是把一些经验概率（分布）乘起来，然后比大小，所以我们就可以通过可视化这些经验分布来完成朴素贝叶斯本身的可视化工作。

相应的实现可以参见 https://github.com/carefree0910/MachineLearning/tree/Book/b_NaiveBayes/Vectorized 中的代码, 这里我们就直接展示这种可视化在 2.2.4 节中用过的蘑菇数据集上的表现。由于蘑菇数据一共有 22 维, 所以可视化经验分布时一共会生成 22 张图, 从这些图中可以非常清晰地看出训练数据集各维度特征的分布, 下面我们就选出几组有代表性的图片进行说明。

一般来说, 一组数据特征中会有相对“重要”的特征和相对“无足轻重”的特征, 通过上述链接中实现的可视化就可以较为轻松地辨析出在离散型朴素贝叶斯中这两者的区别。比如, 在离散型朴素贝叶斯算法中, 相对重要的特征的表现会如图 D.1、D.2 所示。



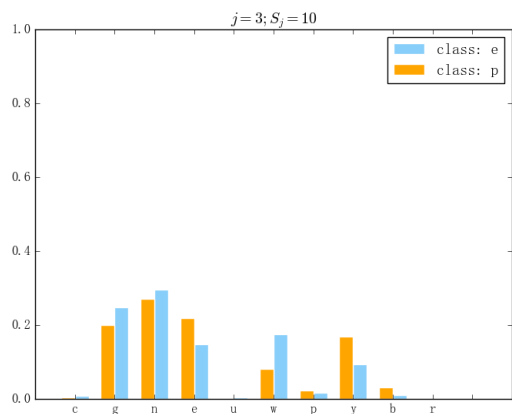
图D.1 蘑菇数据集第5维的特征分布



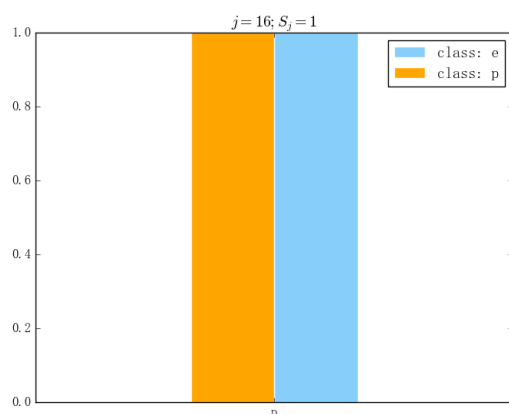
图D.2 蘑菇数据集第19维的特征分布

可以看出, 蘑菇数据集在第 19 维上两个类别各自的“优势特征”都非常明显, 而第 5 维上两个类别各自特征的取值更是基本没有交集。可以想象, 第 5 维特征就是所谓的“关键特征”, 即使我们只根据它的取值来进行类别的判定, 最后的准确率也一定会非常高。这正是为何我们说蘑菇数据集非常简单, 以及为何朴素贝叶斯与决策树在蘑菇数据集上的表现都非常好的原因。事实上在下一节介绍决策树的可视化时, 我们就会看到决策树确实能够将这第 5 维特征挑选出来并作为分类的一大依据。

当然也有与之相对应的、没那么重要的特征, 它们在可视化后也会有比较明显的特性。比如, 在离散型朴素贝叶斯中相对没那么重要的特征的表现会如图 D.3、D.4 所示。



图D.3 蘑菇数据集第3维的特征分布



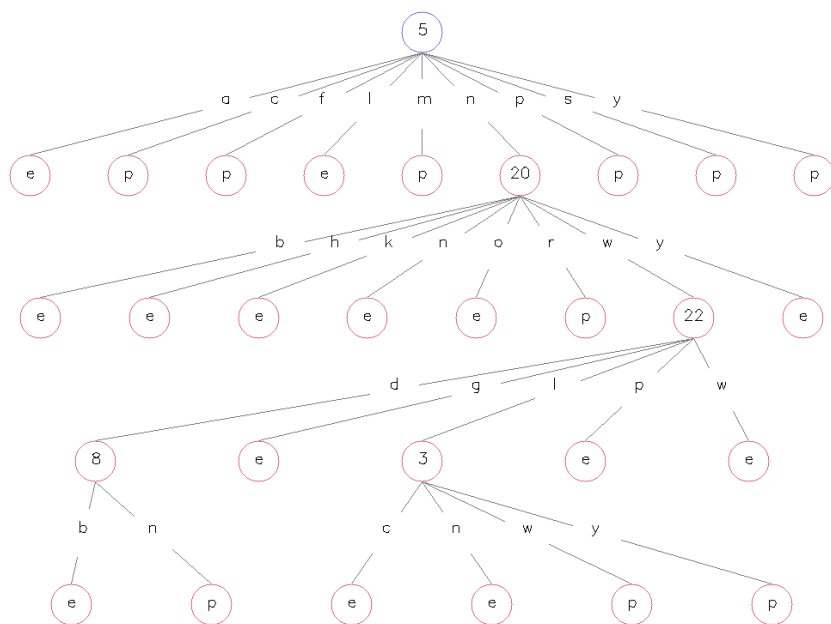
图D.4 蘑菇数据集第16维的特征分布

可以看出，蘑菇数据集在第 3 维上两个类的特征取值基本没有什么差异，第 16 维数据则更是似乎完全没有存在的价值，像这样的特征就可以考虑直接剔除掉。

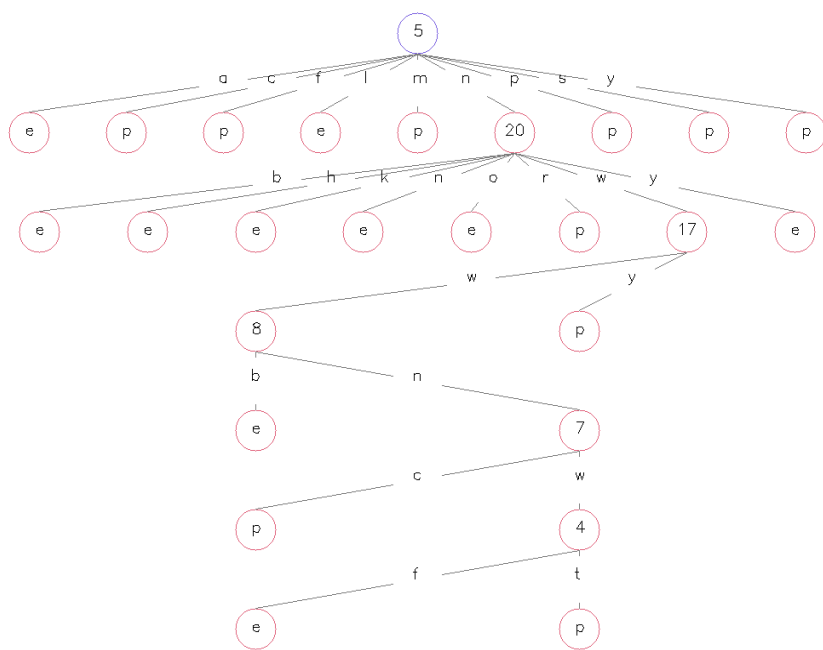
D.2 决策树的可视化

正如第 2 章所说，决策树是从直观上非常好理解的模型，因为它的模型确实就可以用图论中的树来表示，这意味着决策树模型本身是可以被直接可视化出来的（而不是像朴素贝叶斯那样，只能通过可视化各个类别相应的概率分布来间接地可视化朴素贝叶斯本身）。

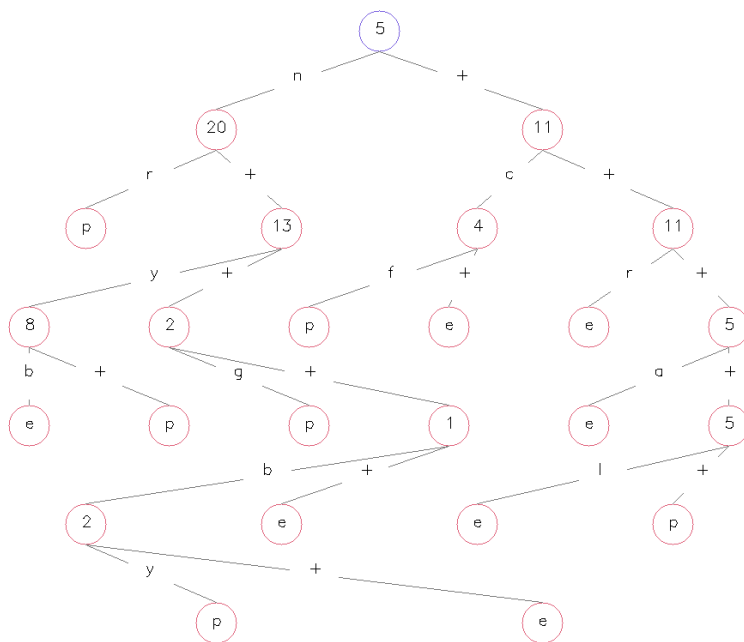
于是，我们可能自然就会希望程序能将生成的决策树画成如图 2.1 所示的简明清晰的模样。虽然用简单的方法不能画得那么漂亮，但是确实是可以做出类似的结构。下面就先展示一下（蘑菇数据集上的）可视化的效果（如图 D.5~D.8 所示），然后再展示相对应的代码。



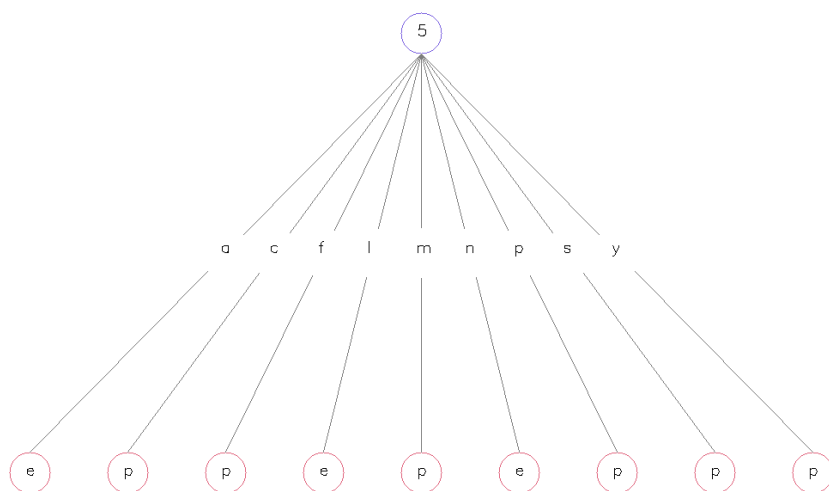
图D.5 蘑菇数据集上ID3决策树的可视化



图D.6 蘑菇数据集上C4.5决策树的可视化



图D.7 蘑菇数据集上CART决策树的可视化



图D.8 蘑菇数据集上单层决策树的可视化

其中，各个数字代表该节点作为划分标准的特征所属的维度，位于各条连线中央的字母代表着该维度特征的各个取值，加号“+”代表着“其他”，各个圆圈中的字母代表类别标记。以上几张可视化的图在一定程度上验证了很多关于决策树的说法，比如 ID3 会倾向选择取值比较多的特征、C4.5 可能会倾向选择取值比较少的特征且倾向于在每个二叉分枝处留下一个小节点作为叶节点、CART 各个叶节点上的样本分布较均匀且生成出的决策树会比较深……同时，这些图也非常清晰地告诉了我们蘑菇数据集中第 5 维特征的重要性，因为无论是哪种决策树模型，根节点处都是第 5 维特征。

下面就来看看这种可视化对应的代码实现，它是笔者用 numpy 进行的决策树内嵌的实现（https://github.com/carefree0910/MachineLearning/tree/Book/c_CvDTree），不过即使单独拿出来也可以大致说明问题（完整的代码可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/c_CvDTree/Tree.py），如代码 D.1 所示。

代码 D.1 决策树可视化的实现

```

01 """
02     可视化决策树
03     radius: 每个节点的半径
04     width、height: 生成的可视化图的宽度、高度
05     height_padding_ratio: 高度方向的衬垫比例
06     width_padding: 宽度方向的衬垫值
07 """
08 def visualize(self, radius=24, width=1200, height=800,
09               height_padding_ratio=0.2, width_padding=30):
10     # self.layers 属性存储着决策树中各层的中间节点
11     # 所以 n_units 代指各层中间节点的个数
12     n_units = [len(layer) for layer in self.layers]
13
14     # 初始化可视化图为全白的一张图
15     img = np.ones((height, width, 3), np.uint8) * 255

```

```

16     # 根据衬垫比例, 算出高度方向的衬垫值
17     height_padding = int(
18         height / (len(self.layers) - 1 + 2 * height_padding_ratio)
19     ) * height_padding_ratio + width_padding
20     # 然后计算出高度方向上, 决策树各层节点所对应的高度
21     height_axis = np.linspace(
22         height_padding, height - height_padding, len(self.layers),
23         dtype=np.int)
24     # 并计算出宽度方向上, 决策树各层中各个节点所对应的宽度
25     width_axis = [
26         np.linspace(width_padding, width - width_padding, unit + 2,
27             dtype=np.int)
28         for unit in n_units
29     ]
30     width_axis = [axis[1:-1] for axis in width_axis]
31
32     # 对这些高度和宽度进行遍历
33     for i, (y, xs) in enumerate(zip(height_axis, width_axis)):
34         for j, x in enumerate(xs):
35             # 根据坐标信息画出各个节点
36             if i == 0:
37                 cv2.circle(img, (x, y), radius, (225, 100, 125), 1)
38             else:
39                 cv2.circle(img, (x, y), radius, (125, 100, 225), 1)
40             # 提取出对应的决策树节点
41             node = self.layers[i][j]
42             # 根据该节点是否是中间节点来进行相应的绘制
43             # 同时, 提取出该节点的特征 (或类别) 信息
44             if node.feature_dim is not None:
45                 text = str(node.feature_dim + 1)
46                 color = (0, 0, 255)
47             else:
48                 text = str(self.y_transformer[node.category])
49                 color = (0, 255, 0)
50             # 把相应的特征 (或类别) 信息输出到可视化图上
51             cv2.putText(
52                 img, text, (x-7*len(text)+2, y+3),
53                 cv2.LINE_AA, 0.6, color, 1)
54
55     # 以上是节点的可视化, 下面就要开始做节点之间的连线的可视化了
56     # 首先仍然是遍历各个节点 (的宽度和高度)
57     for i, y in enumerate(height_axis):
58         if i == len(height_axis) - 1:
59             break
60         for j, x in enumerate(width_axis[i]):
61             new_y = height_axis[i + 1]
62             dy = new_y - y - 2 * radius
63             for k, new_x in enumerate(width_axis[i + 1]):
64                 dx = new_x - x

```

```

65         length = np.sqrt(dx**2+dy**2)
66         ratio = 0.5 - min(0.4, 1.2 * 24/length)
67         # 观察遍历到的两个节点是否存在“父子关系”
68         # 如果存在的话,就把相应的信息输出到可视化图中
69         if self.layers[i + 1][k] in self.layers[i][j].children.values():
70             cv2.line(
71                 img, (x, y+radius),
72                 (x+int(dx*ratio), y+radius+int(dy*ratio)),
73                 (125, 125, 125), 1)
74             cv2.putText(
75                 img, str(self.layers[i+1][k].prev feat),
76                 (x+int(dx*0.5)-6, y+radius+int(dy*0.5)),
77                 cv2.LINE_AA, 0.6, (0, 0, 0), 1)
78             cv2.line(
79                 img, (new x-int(dx*ratio),
80                     new y-radius-int(dy*ratio)),
81                 (new x, new y-radius),
82                 (125, 125, 125), 1)
83
84         # 将可视化图显示出来,并返回该可视化图
85         cv2.imshow(title, img)
86         cv2.waitKey(0)
87         return img

```

以上就是决策树可视化的全部实现,可能需要一些 cv2 的知识才能完全理解,不过大体的思路还是比较清晰的:先计算出各层节点所处的位置,然后用两轮遍历依次将节点和节点之间的连线进行可视化。

附录 E

模型的评估指标

虽然本书介绍了许多算法而且也做了很多实验，但是对于初次接触机器学习，甚至对于接触机器学习已有一定时间的人来说，实验中所用的评估指标究竟是怎么定义的、应该在哪些场景用哪些指标，恐怕是没有完全掌握的。因此，本附录拟介绍几种最常见的模型评估指标的定义与应用场景；与此同时，我们还会给出一种非常有效的、处理二分类问题的方法，利用该方法一般能够在不改变二分类模型的情况下，显著提升二分类模型的相应性能。

本附录将涵盖如下内容：

- acc 的定义与应用场景
- auc 的定义与应用场景
- mse 的定义与应用场景
- corr 的定义与应用场景

同时为了方便讨论，如无特殊说明，本附录将统一使用如下符号范式：

- 假设测试集中的 N 个标签为 y_1, y_2, \dots, y_N ，且：
 - 若问题为二分类问题，则 $y_1 \sim y_N$ 都是 -1 或 1。
 - 若问题为多分类问题（比如说 K 类），则 $y_1 \sim y_N \in \{0, 1, \dots, K-1\}$ 。
 - 若问题为回归问题，则 $y_1 \sim y_N$ 都为（规定范围内的）实数。
- 假设训练好的模型的相应预测标签为 $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_N$ ，且：
 - 问题为二分类问题，则 $\hat{y}_1 \sim \hat{y}_N$ 都是 0 到 1 之间的实数。
 - 若问题为多分类问题（比如说 K 类），则 $y_1 \sim y_N$ 都是（ K 维的）概率向量。
 - 若问题为回归问题，则 $y_1 \sim y_N$ 都为（规定范围内的）实数。

注意：由于相应实现已经在 3.5.2 节的代码 3.2 中给出过，所以本附录就只叙述理论部分而略去具体的代码实现。

E.1 Accuracy (acc)

准确率（Accuracy）是最普适的模型评估指标之一，它不仅能在分类问题中应用，也能在某些回归问题中应用。其中，分类问题中准确率的定义可以说是耳熟能详，而且在 2.1.3 节中也曾有提及：

$$\text{acc} = \begin{cases} \frac{1}{N} \cdot \sum_{i=1}^N I((y_i = 1 \wedge \hat{y}_i \geq \epsilon) \vee (y_i = -1 \wedge \hat{y}_i < \epsilon)) & , \text{ if 二分类} \\ \frac{1}{N} \cdot \sum_{i=1}^N I(\arg \max_k \hat{y}_i = y_i) & , \text{ if 多分类} \end{cases}$$

在上面这个公式中，二分类情形里面的 ϵ 是模型将第 i 个样本归类为正样本（即认为 $y_i = 1$ ）的阈值，一般来说可以简单地取为 0.5，即如果模型在第 i 个样本上输出的值大于 0.5 的话，就认为该样本为正样本。

然而，这种做法在某些情况下是不尽合理的。由于在理想情况下，我们希望模型输出的是样本属于正样本的概率值，所以只有在正负样本均衡的情况下，取 0.5 作为阈值才是一个合乎逻辑的选择；假设原数据集中正负样本极不均衡的话，那么还取 0.5 作为阈值就显得有失偏颇。这时，一个朴素的解决方案就是用类别的先验概率来作为阈值。比如，如果正样本量：负样本量 = 1 : 9 的话，那么取 0.9 作为阈值就是一个不错的选择。

当然，确实还有更加合理、从数学上来看也更加优雅的方法来挑选二分类的阈值 ϵ ，我们会在下一节介绍完 auc 之后补上相应的说明。

至于如何在回归问题中应用 acc，常见的做法就是先将回归问题转化为分类问题，然后再用分类问题中 acc 的定义来加以应用。一个常见的应用场景就是股票预测，理想的模型当然是直接用模型回归出股票将来的价格，但是即使将问题简化成预测股票未来的涨跌（即将“预测价格”这个回归问题转化为“预测涨跌”这个二分类问题），其实也是很困难而且很值得做的任务，此时我们就能使用 acc 来判断模型的“胜率”了。

E.2 Area Under the Curve (auc)

准确率的泛用性确实很强，但是仍然有一些不能用 acc 作为评估指标的场景，比如上一节提到过的类别不均衡的场景。试想如果正负样本之比达到了 1 : 99 的话，毫无疑问正样本才是最关键的样本，我们本应尽可能地将正样本找出来。但是，即使我们将全部样本都预测为负样本，准确率也有 0.99。这就意味着即使我们完全忽视掉最重要的东西，我们的模型从准确率这个指标看仍然是非常好的模型。

为了解决这个矛盾，auc 指标就应运而生了。需要指出的是，auc 在代指模型评估指标时，大多数都是“auroc (Area Under the Receiver Operating Characteristic curve)”的简写。换句话说，auc 中的 Curve 在代指模型评估指标时，大多数情况下都是 Receiver Operating Characteristic curve (ROC curve)。

由于这里面涉及的数学知识不少，而且它们与本书主题的关系也不大，所以我们就着重用直观的语言去说明一下这些名词的含义。首先，在 auc 指标之前，还有一个更好理解的、能够处理类别不均衡场景的指标，叫作 F_1 -score。仍然用回之前正负样本之比达到了 1 : 99 的例子，此时我们会很希望将正样本找出来，即我们很希望将正样本“召回”。但是，这不意味着我们应该盲目地“召回”正样本，还需要保证“召回”的正样本都是正样本，即应该保证召回的正样本的“精确度”尽可能高。

注意：这里的精确度和上一节介绍的准确率稍有不同：精确度特指召回的正样本的准确率，而上一节介绍的准确率则泛指所有样本的准确率。

在这种思想下，我们就可以很自然地定义出两个指标：召回率(recall)和精确度(precision)。为方便讨论，我们先用阈值来将模型的预测值转化为二值预测，即：

$$\hat{y}_i \leftarrow \begin{cases} 1 & , \text{ if } \hat{y}_i \geq \epsilon \\ -1 & , \text{ if } \hat{y}_i < \epsilon \end{cases}$$

那么此时的 recall 和 precision 的定义即为：

$$\begin{cases} \text{recall} &= \frac{tp}{tp + fn} \\ \text{precision} &= \frac{tp}{tp + fp} \end{cases}$$

其中， tp 、 fp 、 fn 的定义如图 E.1 所示。

	(模型预测为) 正样本	(模型预测为) 负样本
(真实标签为) 正样本	tp	fn
(真实标签为) 负样本	fp	tn

其中， tp 、 fp 、 tn 、 fn 分别是 True Positive、False Positive、True Negative、False Negative 的缩写。

图E.1 tp 、 fp 、 tn 与 fn 的定义

不难看出， $tp + fn$ 其实就是所有的正样本，所以 recall 就是模型找到的正样本量 (tp) 占所有正样本量的比值，即直观意义上的“召回率”。而 $tp + fp$ 其实就是模型认为是正样本的数量，所以 precision 就是模型找到的、真正的正样本量 (tp) 占所有模型认为是正样本的比值，即直观意义上的“精确度”。

那么，既然我们要同时考虑这两个因素的话，一个直观的想法就是取它们的调和平均：

$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = \frac{2 \cdot tp}{2 \cdot tp + fn + fp}$$

这样的话,当 recall 和 precision 都达到最大值 1 时, F_1 就会是 1; 而如果其中一个特别差(即特别接近于 0)的话, F_1 就会特别接近 0。此外,只要 recall 或者 precision 在另外的指标不变下变高,那么 F_1 就一定也会随着变高。总而言之, F_1 是一个综合考虑了 recall 和 precision 的、简单易算且有效的模型评估指标,在很多不均衡数据集上都有应用。

不过,虽然 F_1 的实用性确实很强,但它毕竟还是不够有针对性。具体而言,它将 recall 和 precision 视为同等重要的指标,这在某些特定的任务中是不尽合理的。为了解决这个问题,我们就可以用 F_β -score 这个 F_1 -score 的直接变体:

$$F_\beta = \left(1 + \frac{1}{\beta^2}\right) \cdot \frac{1}{\frac{1}{\text{recall}} + \frac{1}{\beta^2 \cdot \text{precision}}} = \frac{(1 + \beta^2) \cdot tp}{(1 + \beta^2) \cdot tp + \beta^2 \cdot fn + fp}$$

不难看出,当 $\beta = 1$ 时, F_β 其实就是 F_1 。常用的两种指标为 $F_{0.5}$ ($\beta = 0.5$) 和 F_2 ($\beta = 2$),其中前者认为 precision 更重要,后者认为 recall 更重要。

如果能够把 F_1 和 F_β 理解透彻的话,相信大家就已经对 tp 、 fp 、 tn 与 fn 这四个定义有比较深刻的认知了,于是我们就可以来看看本节的主题——auc 的定义了。由前文可知, auc 的定义为 Area Under the ROC Curve,所以自然就需要先知道何为 ROC 曲线。大体上来说, ROC 曲线在统计学中代指一个能够反映二分类模型在不同阈值 ϵ 下的表现;具体而言,它会画出二分类模型在不同 ϵ 下的 True Positive Rate (TPR) 和 False Positive Rate (FPR) 所组成的曲线。其中,它会以 TPR 为 y 轴, FPR 为 x 轴,且:

$$TPR = \frac{tp}{tp + fn}, \quad FPR = \frac{fp}{fp + tn}$$

换句话说, TPR 是预测正确的正样本占有所有正样本的比例,而 FPR 是预测错误的负样本占有所有负样本的比例,所以我们自然希望 TPR 更大、FPR 更小,这和我们之前希望 recall、precision 都尽可能大是差不多的。但是, auc 与 F_1 、 F_β 最不同的也是最有优势的一点就是,它并不是简单地把这两项指标进行某种组合,而是计算这两项指标更具有统计意义的统计量:它会将计算“ROC 曲线下的区域”作为最终的衡量目标。如果用没那么严谨的数学语言来叙述的话,由于 ROC 曲线分别以 TPR、FPR 为 y 轴、x 轴,不妨假设 ROC 曲线能够用一个可积函数 f 来表示出来(虽然这个假设不易证明且不总是正确,但直观上是可以接受的,因为一个模型如果足够稳定的话,只要阈值 ϵ 的变化不大,那么 TPR、FPR 的变化也应该一致地不大)。

注意: 虽然由于在不同 ϵ 下可能会出现 FPR 相同但 TPR 不同的情形(即 x 相同但 y 不同的情形),从而导致 ROC 曲线严格意义上来说不能是一个函数的图像,但如果我们只保留 ROC 曲线上同一个 x 下最大的 y 的话, ROC 曲线在大部分情况下就可以用可积函数 f 来表示了。

此时, auc 的定义其实非常简洁明了:

$$\text{auc} = \int_0^1 f(x) dx$$

即 auc 就是 ROC 曲线对应的可积函数 f 在定义域上的黎曼积分。那么这个定义意味着什么呢？首先由 TPR 和 FPR 的性质不难看出，ROC 曲线和 f 拥有如下三条性质：

- 定义域、值域都是 0 到 1（即 $x, y \in [0, 1]$ ），这是因为 TPR 、 FPR 的取值范围都是 0 到 1。
- 原点位于 ROC 曲线上（但不一定有 $f(0) = 0$ ），这是因为当阈值 $\epsilon = 1$ 时所有样本都会被预测为负样本，所以自然就有

$$tp = fp = 0$$

从而就有

$$TPR = FPR = 0$$

- $(1, 1)$ 位于 ROC 曲线上（从而一定有 $f(1) = 1$ ），这是因为当阈值 $\epsilon = 0$ 时所有样本都会被预测为正样本，所以自然就有

$$fn = tn = 0$$

从而就有

$$TPR = FPR = 1$$

注意，我们希望 $TPR(y)$ 尽可能大、 $FPR(x)$ 尽可能小，所以最理想的情况下，函数 f 的表达式应为

$$f(x) = 1, \quad x \in [0, 1]$$

这意味着当阈值 ϵ 从 1 开始逐渐减小时，模型能在保持 FPR 为 0 的同时慢慢地提高 TPR ，直到 TPR 变成（理想的）1，此时模型就同时做到了“预测对所有的正样本”和“没有预测错任何的负样本”，即此时的模型是最理想的模型。从此往后，当我们再继续减小 ϵ 时，就会将一些本来预测对的负样本预测成正样本，从而使得 FPR 慢慢上升，最终到达 1。在此过程中，由于原来预测为正样本的样本依然会被预测为正样本，所以 TPR 会一直保持在 1。

由前文可知， f 的定义域和值域都是 $[0, 1]$ ，所以当 $f(x) \equiv 1$ 时， $f(x)$ 在 0 到 1 上的积分就是 1。所以在最理想的情况下，我们会有 $auc = 1$ 。

注意： auc 的更多（概率）内涵可参见 <https://stats.stackexchange.com/questions/180638/how-to-derive-the-probabilistic-interpretation-of-the-auc>。

综上所述，auc 指标的取值范围与其余指标（acc、 F_β ）一致（都是 $[0, 1]$ ），但它的评估思想却和 acc、 F_β 这类指标很不相同。后者需要先定出具体的类别之后再计算，这就要求我们人为地提供一个阈值 ϵ ，从而带来一定的偏差；但前者会综合考虑所有 ϵ 的情况，从统计意义上来看会合理得多。

此外，auc 是可以用来评估多分类模型的（对应代码 3.2 中的 multi_auc 方法），大致思想类似于 One-vs.-rest（https://en.wikipedia.org/wiki/Multiclass_classification#One-vs.-rest），详情可参见 <https://stats.stackexchange.com/questions/2151/how-to-plot-roc-curves-in-multiclass-classification>。

在本节的最后，我们来看看如何利用 auc 指标来选取二分类模型的阈值 ϵ 。由于 ROC 曲线的绘制需要算出所有阈值 ϵ 下的 TPR 和 FPR ，所以如果我们想要最大化模型在另一个指标下的表现的话，只要该指标能被 TPR 和 FPR 表示出来，那么我们自然就能选出该指标下的最优阈值 ϵ 。以 acc 为例，由于

$$\text{acc} = \frac{tp + tn}{tp + fp + tn + fn}$$

那么如果令 pos 为正样本所占的比例，即

$$\text{pos} = \frac{tp + fn}{tp + fp + tn + fn}$$

就会有

$$\begin{aligned}\text{acc} &= \frac{tp}{tp + fp + tn + fn} + \frac{tn}{tp + fp + tn + fn} \\ &= \frac{tp}{tp + fn} \cdot \frac{tp + fn}{tp + fp + tn + fn} + \frac{tn}{tn + fp} \cdot \frac{tn + fp}{tp + fp + tn + fn} \\ &= \text{TPR} \cdot \text{pos} + (1 - \text{FPR}) \cdot (1 - \text{pos})\end{aligned}$$

即 acc 确实能被 TPR 和 FPR （和一个不随阈值 ϵ 改变而改变的常量 pos ）给表示出来，从而我们就能在计算完 auc 之后，自然地选出使得（二分类）模型 acc 最大的阈值 ϵ 。

E.3 Mean Squared Error (mse)

前两节我们介绍了三种常用的分类模型的评估指标（ acc 、 F_β 和 auc ），这一节和下一节我们则将会介绍两种常用的回归模型的评估指标。其中，本节将介绍的 mse 指标其实也可以用于评估分类模型，只不过 mse 本身不具有概率意义，这对于输出通常为概率输出的分类模型而言是非常别扭的。所以在本节有限的讨论中，我们默认使用的是回归模型。

mse 的定义在第 3 章介绍损失函数时曾说过，只不过彼时我们将 mse 视为损失函数，而在这里我们将 mse 视为模型评估指标：

$$\text{mse} = \frac{1}{N} \cdot \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

也就是说，当回归模型的预测值 \hat{y}_i 和真实标签 y_i 越接近时， mse 的值就会越小。在最理想的情况下，由于总有 $\hat{y}_i = y_i (i = 1 \sim N)$ ，所以此时就有

$$\text{mse} = 0$$

即最理想的 mse 就是 0。从这里也可以看出，与前两节介绍的所有指标不同， mse 是越小越好的指标。

不过，虽然 mse 能够告诉我们模型的预测值与真实标签之间的距离，但是这个距离究竟算好还是坏却不得而知。一种常用的比较手段是拿它和“方差”的极大似然估计相比，即计算

$$\sigma^2 \triangleq \frac{1}{N} \cdot \sum_{i=1}^N (y_i - \bar{y})^2$$

其中， \bar{y} 是训练集上真实标签的均值。然后，我们就可以通过比较 mse 和 σ^2 的大小来判定当前模型（在以 mse 为指标时）的好坏，这是因为当我们简单粗暴地使用训练集标签均值作为模型的输出时的 mse 指标即为 σ^2 ，所以一个好的模型至少应该比这种输出恒定数值的模型要更好才行。

E.4 Correlation (corr)

上一节介绍的 mse 指标和第 3 章介绍的 mse 损失函数一样，都属于“万金油”：它能在所有场合进行运用，但也几乎无法在任何场合做到最好。具体而言，由于 mse 指标希望我们将预测值和真实标签“尽可能地接近”，所以它就无法处理这样一类回归问题：只希望预测值和真实标签“尽可能相关”。一个典型的例子就是销量预测，如果我们只是想预测出“销量最好的五个商品”而不是“每个商品的销量的话”，那么我们其实只用给每个商品打分然后选出分数最高的五个商品，同时保证这个分数和销量正相关即可，这个任务往往会比精准地预测出每个商品的销量要简单不少。

相关性的定义也有很多种，这里就给出其中颇具盛名的 Pearson correlation coefficient 的定义，我们一般说起“相关系数”时说的也正是这种定义：

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y}$$

其中， cov 代指协方差， σ 代指相应随机变量 (X, Y) 的标准差。值得一提的是，与 mse 类似， corr 不仅能够作为评估指标来使用，还可以作为损失函数来使用。我们在第 3 章中同样有相应的代码实现，这里不再赘述。

附录 F

实现补足

本附录会补充说明本书正文中用到过但却和神经网络本身关系不大的代码实现，它们大多是各种 Python、numpy、TensorFlow 技巧的运用。具体而言，本附录将会涵盖如下内容：

- 数据生成器的实现（出现于 3.5.2 节）
- 利用数据计算 TensorFlow 中的 Tensor 的方法（出现于 3.5.3 节）
- 提取数据信息的方法（出现于 5.2.3、6.1.2、6.2.2 节）
- DataCacheMixin 的实现（出现 7.1 节）

F.1 数据生成器的实现

数据生成器是我们为了让 TensorFlow 基础模型（Base）适应于各种场景所做的抽象，它能够将数据获取与模型算法这两大部分内容分离开来。虽然本书用到的数据生成器的功能非常简单（只用生成一个个 Mini Batch 即可），但在此基础上进行拓展是非常自然的，大家完全可以根据自己的需要来定制属于自己的数据生成器。笔者实现的、比较完整的数据生成器可以参见 https://github.com/carefree0910/MachineLearning/blob/Book/_Dist/NeuralNetworks/Base.py，这里面不仅涵盖了 DNN 所用的数据生成器（Generator），也涵盖了 RNN 所用的数据生成器（Generator3D）和 CNN 所用的数据生成器（Generator4D）。不过囿于篇幅，在这里我们还是只展示 Generator 中的核心实现。首先来看看它的初始化，如代码 F.1 所示。

代码 F.1 Generator 的实现

```
01 """
02     数据生成器
03     x、y：特征向量与标签
04     name：数据生成器的名字
```

```

05     weights: 数据集中各样本的权重
06     n class: 类别数; 如果是回归问题的话, 就会有n_class = 1
07     shuffle: 是否打乱数据集
08     """
09     class Generator:
10         def init (self, x, y, name="Generator",
11                 weights=None, n class=None, shuffle=True):
12             # 定义一个储存缓存的属性
13             self.cache = {}
14             # 将输入的特征向量和标签都转为 numpy 数组
15             self.x, self.y = np.asarray(x, np.float32), np.asarray(y, np.float32)
16             # 如果提供了样本权重的话, 就转成 numpy 数组并进行记录
17             if weights is None:
18                 self.sample weights = None
19             else:
20                 self.sample weights = np.asarray(weights, np.float32)
21             # 如果提供了类别数的话就直接记录
22             if n class is not None:
23                 self.n class = n class
24             # 否则, 就根据提供的标签来算出类别数
25             else:
26                 y int = self.y.astype(np.int32)
27                 if np.allclose(self.y, y int):
28                     # 注意如果是分类问题的话, 我们会要求类别从 0 开始
29                     assert y int.min() == 0, "Labels should start from 0"
30                     self.n class = y int.max() + 1
31                 else:
32                     self.n class = 1
33             # 初始化各项属性
34             self.name = name
35             self.do shuffle = shuffle
36             # 根据 self.generate all valid data 方法, 获取所有“有效的”数据
37             self.all valid data = self.generate all valid data()
38             # 用 self._valid_indices 属性, 记录所有有效数据的下标
39             # 在最简单的情况下, 所有数据都是有效的
40             self._valid_indices = np.arange(len(self._all_valid_data))
41             # 用 self._random_indices 属性, 记录打乱后有效数据的下标
42             self._random_indices = self._valid_indices.copy()
43             np.random.shuffle(self._random_indices)
44             # 用 self._batch_cursor 属性, 记录当前 Mini Batch 的下标
45             self._batch_cursor = -1
46
47             # 定义一个能返回有效数据量的 property
48             @property
49             def n_valid(self):
50                 return len(self._valid_indices)
51
52             # 定义一个能获取特征向量维度的 property
53             @property

```



```

54     def n_dim(self):
55         return self.x.shape[-1]
56
57     # 定义一个能获取特征向量的 shape 的 property
58     @property
59     def shape(self):
60         return self.n_valid, self.n_dim

```

其中在最简单的情况下，`self.generate_all_valid_data` 方法就只是简单地把特征向量和标签并在一起而已：

```

61     def _generate_all_valid_data(self):
62         return np.hstack([self.x, self.y.reshape([-1, 1])])

```

在完成了初始化之后，我们就可以实现核心的、随机生成 Mini Batch 的方法了。需要指出的是，虽然确实可以将生成 Mini Batch 的过程完全随机化，但是在实际任务中，为了利用上所有的数据集，我们常常会通过先将数据集打乱，然后按顺序来生成 Mini Batch 的方式来完成“伪随机化”，其核心的代码实现如下所示。

```

63     # 参数 n_batch 代指 Mini Batch 中的样本量
64     # 参数 re_shuffle 代指是否在依次生成完所有数据后重新打乱数据
65     def gen_batch(self, n_batch, re_shuffle=True):
66         # 如果 n_batch 比所有有效样本量还大的话
67         # 就需要将 n_batch 调整成所有有效样本量
68         n_batch = min(n_batch, self.n_valid)
69         # 如果 n_batch 是-1的话，就认为需要将所有有效数据都生成出来
70         if n_batch == -1:
71             n_batch = self.n_valid
72         # 如果 self.batch_cursor 属性小于 0，说明现在处于初始状态
73         # 从而要把 self.batch_cursor 属性设置为 0
74         if self.batch_cursor < 0:
75             self.batch_cursor = 0
76         # 如果需要打乱数据的话
77         if self.do_shuffle:
78             # 而且如果处于初始状态且需要重新打乱数据的话
79             if self.batch_cursor == 0 and re_shuffle:
80                 # 就重新打乱数据的下标
81                 np.random.shuffle(self.random_indices)
82                 indices = self.random_indices
83             # 否则，直接用 self.valid_indices 来获取数据下标即可
84         else:
85             indices = self.valid_indices
86         # 定义一个变量来标识当前数据是否会被生成完
87         end = False
88         # 定义一个变量来记录下一次生成 Mini Batch 时的下标
89         next_cursor = self.batch_cursor + n_batch
90         # 如果下一次的下标超过了有效样本量
91         if next_cursor >= self.n_valid:
92             # 就将下一次的下标设置为有效样本量

```

```

93         next_cursor = self.n valid
94         # 并标识“当前数据已经生成完”
95         end = True
96         # 根据 self.get_data 方法, 获取 Mini Batch 及相应的样本权重
97         data, w = self.get_data(indices[self.batch_cursor:next_cursor])
98         # 如果已经生成完所有数据, 就将 self.batch_cursor 重置为-1
99         # 否则, 就设置为下一次的下标
100        self.batch_cursor = -1 if end else next_cursor
101        # 返回 Mini Batch 及相应的样本权重
102        return data, w

```

其中, 最简单情况下的 `self.get_data` 方法的实现如下:

```

103    def get_data(self, indices, return_weights=True):
104        data = self.all_valid_data[indices]
105        if not return_weights:
106            return data
107        if self.sample_weights is None:
108            weights = None
109        else:
110            weights = self.sample_weights[indices]
111        return data, weights

```

在这种抽象下, 如果实际的任务更为复杂(比如, 如果数据集中蕴含着时间序列信息的话, 对于 DNN 而言就需要将连续的若干个样本“拼接”成一个样本), 我们只需要将上述对应的方法进行更改即可, 大体的框架则无须做出任何改动。

F.2 TensorFlow Tensor 的计算

在 TensorFlow 中, 我们会需要将所有的变量及相应的运算都写进底层的运算图 Graph 中, 这虽然大大提高了程序运行的速度, 但也带来了不少麻烦。具体而言, 在平常的编程中, 如果我们想要知道某个变量的值, 只需简单地将其 print 出来或 pickle 在二进制文件中即可; 但是对于 TensorFlow 而言, 如果我们想要知道 Graph 中某个张量 (Tensor) 的具体取值的话, 由于该取值通常取决于具体的输入数据, 所以我们就必须根据数据来计算出这个 Tensor。然而不幸的是, 在实际任务中, 由于数据量通常过于庞大, 所以很有可能出现内存放不下所有数据的情况, 此时我们就还是需要先将数据分成若干 Mini Batch, 然后算出各个 Mini Batch 下 Tensor 的取值, 最后再把这些取值拼接起来作为最终结果。本节所要介绍的就是这个过程的实现, 如代码 F.2 所示。

注意: 以下介绍的 (`_calculate`) 方法是从属于第 3 章介绍过的 TensorFlow 基本框架 (Base) 的方法, 我们在 3.5.3 节中首次运用到了它。同时需要注意的是, 由于 Tensor 在 Base 中都是以类属性的形式存在的, 所以该方法的本质是利用数据来计算各个属性。

代码 F.2 _calculate 方法的实现

```

01     """
02     计算 Base 及其衍生模型的各个属性（所对应的 Tensor）
03     x、y: 计算 Tensor 时用到的特征向量与标签
04     weights: 各样本的样本权重
05     tensor: 需要计算的 Tensor
06     n_elem: 一个 Mini Batch 中的最大数字个数
07     is_training: 标识着现在是否正处于训练过程
08     """
09     def _calculate(self, x, y=None, weights=None,
10                   tensor=None, n_elem=1e7, is_training=False):
11         # 根据特征维度和 n_elem, 计算出一个 Mini Batch 中的样本量
12         n_batch = int(n_elem / x.shape[1])
13         # 计算出总共需要对多少个 Mini Batch 进行 (Tensor 的) 计算
14         n_repeat = int(len(x) / n_batch)
15         if n_repeat * n_batch < len(x):
16             n_repeat += 1
17         # 定义一个变量, 储存各个属性所对应的下标
18         cursors = [0]
19         # 默认计算模型概率输出所对应的属性 (self._prob_output)
20         if tensor is None:
21             target = self._prob_output
22         # 如果传进来的 Tensor 是一个 list 的话
23         # 就说明要同时计算多个属性 (所对应的 Tensor)
24         elif isinstance(tensor, list):
25             # 先定义一个储存所有要计算的 Tensor 的 list
26             target = []
27             # 遍历所有要计算的 Tensor
28             for t in tensor:
29                 # 如果传进来了 str 的话, 就用 getattr 方法获取相应的属性
30                 if isinstance(t, str):
31                     t = getattr(self, t)
32                 # 如果要计算的属性是一个 Tensor 组成的 list 的话
33                 if isinstance(t, list):
34                     # 就将这些 Tensor 加入 target
35                     target += t
36                     # 同时更新各个属性所对应的下标
37                     cursors.append(len(t))
38                 # 否则, 就只用往 target 中加一个 Tensor
39                 # 下标也只需往前挪一位
40             else:
41                 target.append(t)
42                 cursors.append(cursors[-1] + 1)
43         # 否则, 就说明只需计算一个属性 (所对应的 Tensor)
44         else:
45             target = getattr(self, tensor) if isinstance(tensor, str) else tensor
46         # (利用数据) 计算出所有要计算的 Tensor 的数值
47         results = [self._sess.run(
48             target, self._get_feed_dict(

```

```

49         x[i * n batch:(i + 1) * n batch],
50         None if y is None else y[i * n batch:(i + 1) * n batch],
51         None if weights is None else weights[i * n batch:(i + 1) * n batch],
52         is training=is training
53     )
54 ) for i in range(n repeat)]
55 # 如果 target 不是一个 list 的话
56 # 就只用返回一个属性对应的 (Tensor 的) 数值
57 if not isinstance(target, list):
58     if len(results) == 1:
59         return results[0]
60     return np.vstack(results)
61 # 否则, 如果计算了多个 Mini Batch 的话
62 # 就将各个 Mini Batch 的结果进行汇总
63 if n repeat > 1:
64     results = [
65         np.vstack([result[i] for result in results])
66         if target[i].shape.ndims else
67         np.mean([result[i] for result in results]) for i in range(len(target))
68     ]
69 # 如果只计算了一个 Mini Batch, 自然就只用关注第一个结果
70 else:
71     results = results[0]
72 # 根据各属性对应的下标
73 # 将属性各自对应的 Tensor (的取值) 分别返回
74 if len(cursors) == 1:
75     return results
76 return [results[cursor:cursors[i + 1]] for i, cursor in enumerate(cursors[:-1])]

```

为了使实现更高效, 笔者牺牲了一定的代码可读性并在上述实现中用了较多的列表表达式, 所以理解起来可能会有一些难度。

F.3 数据信息的提取

在得到一个新的数据集之后, 我们首先需要做的, 可能就是对其特征向量与标签进行一定的挖掘以建立机器学习模型。在正文的第 5 章和第 6 章中, 我们定义了 `numerical_idx` 和 `categorical_columns` 这两个属性来存储这些信息, 不过彼时我们都默认了这两个属性可以自动获取就没有给出具体的实现, 本节我们则会补充上相应的代码。

为方便大家理解, 我们先来回顾一下这两个属性的定义。首先是 `self.numerical_idx` 这个属性, 它储存的元素为 `True` 或者 `False`, 其中:

- 第 i 个元素是 `True` 则意味着第 i 个特征是连续型特征。
- 第 i 个元素是 `False` 则意味着第 i 个特征是离散型特征。

此外, `self.numerical_idx` 的最后一个元素将会是标签的类型。具体而言:

- `self.numerical_idx[-1]` 为 `True` 意味着标签是连续的，从而目标就是解决回归问题。
- `self.numerical_idx[-1]` 为 `False` 意味着标签是离散的，从而目标就是解决分类问题。

然后是 `self.categorical_columns` 这个属性，它储存的元素是一个元组 (i, n) ，其中：

- i 意味着第 i 个特征是离散型特征。
- n 意味着该离散型特征有 n 个取值。

除了这两个属性以外，由第 6 章的讨论可知，我们还需要提取出如下三个信息：

- 记录各维特征的所有特征取值的 `list` (`self.feature_sets`)。具体而言，假设第 i 维的特征有 a_1, a_2, \dots, a_{n_i} 这 n_i 个取值的话，那么 `self.feature_sets` 这个 `list` 的第 i 个元素就是含 $a_1 \sim a_{n_i}$ 的一个 Python 集合 (`set`)。而如果第 i 维的特征是冗余特征或连续型特征的话，那么相应的第 i 个元素就是一个空集合。
- 记录各维特征的特征取值个数的 `list` (`self.n_features`)，它能由 `self.feature_sets` 直接推得（比如在上面这个例子中，`self.n_features` 的第 i 个元素就是 n_i ）。
- 记录各维特征是否是数值型特征的 `list` (`self.all_num_idx`)。如果某个维度的特征不是数值型特征的话，就需要先将其转换成数值特征，然后再做后续的处理。

提取出这些信息的方法不止一种，在此就展示一下笔者个人的实现，如代码 F.3 所示。

注意：以下介绍的 (`get_feature_info`) 方法是正文中常常用到的 `Toolbox` 类的方法。

代码 F.3 `get_feature_info` 方法的实现

```

01     """
02     提取数据的信息
03     data: (目标) 数据
04     numerical_idx: 与前文所说的 numerical_idx 意义一致
05     is_regression: 标识当前数据对应的问题是否是回归问题的参数
06     logger: 用于进行日志记录的 logger
07     """
08     @staticmethod
09     def get_feature_info(data, numerical_idx, is_regression, logger=None):
10         # 如果没有提供 numerical_idx 的话
11         # 就说明要用程序来生成 numerical_idx
12         generate_numerical_idx = False
13         if numerical_idx is None:
14             generate_numerical_idx = True
15             numerical_idx = [False] * len(data[0])
16         else:
17             numerical_idx = list(numerical_idx)
18         # 定义“数据的转置”，从而一个特征就对应着一行
19         data_t = data.T if isinstance(data, np.ndarray) else list(zip(*data))
20         # 如果数据类型不是 str 的话，那么就需要考虑存在缺失值 nan 的情形
21         # 这是因为在 Python 中，nan 与 nan 之间会被视为不同的元素
22         # 所以我们在统计特征取值个数时，如果不单独考虑 nan 的话
23         # 就会出很大的问题。因此，此时就需要用 shrink_nan 方法

```

```

24     # 将 nan 的个数缩减为 1 个（否则 nan 就会被重复计数）
25     if type(data[0][0]) is not str:
26         shrink features = [Toolbox.shrink nan(feats) for feat in data t]
27     else:
28         shrink features = data t
29     # 利用 Python 自带的集合（set）数据结构
30     # 来定义 feature sets 与 n features，它们的意义与上文一致
31     feature sets = [
32         set() if idx is None or idx else set(shrink feature)
33         for idx, shrink feature in zip(numerical idx, shrink features)
34     ]
35     n features = [len(feature set) for feature set in feature sets]
36     # 利用 feature sets 和 is number 方法，计算出 all num idx
37     all num idx = [
38         True if not feature set else all(
39             Toolbox.is number(str(feats)) for feat in feature set
40         ) for feature set in feature sets
41     ]
42     # 如果要生成 numerical idx 的话
43     if generate numerical idx:
44         # 就先生成 numpy 数组版本的 shrink features
45         np shrink features = [
46             shrink feature if not all num else
47             np.asarray(shrink feature, np.float32)
48             for all num, shrink feature in zip(
49                 all num idx, shrink features)
50         ]
51         # 然后根据它来看
52         # 是否存在特征类型为整数类型且特征取值各不相同的特征
53         # 如果存在的话，说明该特征很有可能是 id，所以就是冗余特征
54         all unique idx = [
55             len(feature set) == len(np shrink feature) and (
56                 not all num or np.allclose(
57                     np_shrink_feature,
58                     np_shrink_feature.astype(np.int32)))
59             for all_num, feature_set, np_shrink_feature in zip(
60                 all_num_idx, feature_sets, np_shrink_features)
61         ]
62         # 利用 get_numerical_idx 方法与所有的已有信息
63         # 来生成 numerical_idx
64         numerical_idx = Toolbox.get_numerical_idx(
65             feature_sets, all_num_idx, all_unique_idx, logger)
66         # 如果有冗余特征，就将 all_num_idx 相应位置的元素置为 None
67         for i, numerical in enumerate(numerical_idx):
68             if numerical is None:
69                 all_num_idx[i] = None
70     # 如果用户已经提供了 numerical_idx 的话
71     else:
72         for i, (feature_set, shrink_feature) in enumerate(

```

```

73         zip(feature_sets, shrink_features)):
74         if i == len(numerical_idx) - 1 or numerical_idx[i] is None:
75             Continue
76         # 如果某个特征为离散型特征
77         if feature_set:
78             # 而且特征取值个数为 1 的话, 说明对应特征为冗余特征
79             if len(feature_set) == 1:
80                 Toolbox.warn_all_same(i, logger)
81                 all_num_idx[i] = numerical_idx[i] = None
82             continue
83         # 如果某个特征的所有特征都取同一个值的话
84         # 说明该特征为冗余特征
85         if Toolbox.all_same(shrink_feature):
86             Toolbox.warn_all_same(i, logger)
87             all_num_idx[i] = numerical_idx[i] = None
88         # 否则, 若该特征为整数型特征, 而且取值各不相同的话
89         # 说明该特征很有可能是 id, 所以也是冗余的
90         elif numerical_idx[i]:
91             shrink_feature = np.asarray(shrink_feature, np.float32)
92             if np.max(shrink_feature[~np.isnan(
93                 shrink_feature)]) < 2 ** 30:
94                 if np.allclose(
95                     shrink_feature,
96                     np.array(shrink_feature, np.int32)
97                 ):
98                     if Toolbox.all_unique(shrink_feature):
99                         Toolbox.warn_all_unique(i, logger)
100                         all_num_idx[i] = numerical_idx[i] = None
101         # 如果指定了是回归问题的话, 就要更改一些自动生成的信息
102         if is_regression:
103             all_num_idx[-1] = numerical_idx[-1] = True
104             feature_sets[-1] = set()
105             n_features.pop()
106         # 返回所有信息
107         return feature_sets, n_features, all_num_idx, numerical_idx

```

总体来说就是 Python、numpy 各种技巧的运用, 而且代码风格也只是笔者个人的风格而已, 仅做参考即可。

F.4 DataCacheMixin 的实现

本节将会介绍 DataCacheMixin 这个类的实现。由于相应的实现已经在正文中有所说明, DataCacheMixin 本身只是做了一个简单的封装, 所以理解起来应该不算困难, 如代码 F.4 所示。

代码 F.4 DataCacheMixin 的实现

```

01 class DataCacheMixin:
02     # 定义能够获取数据文件夹所在路径的 property

```

```
03     @property
04     def data_folder(self):
05         return self.data_info.get("data_folder", "Data")
06
07     # 定义能够获取临时数据文件夹所在路径的 property
08     @property
09     def data_cache_folder(self):
10         folder = os.path.join(self.data_folder, "Cache", self.name)
11         if not os.path.isdir(folder):
12             os.makedirs(folder)
13         return folder
14
15     # 定义存储数据信息所在文件夹的 property
16     @property
17     def data_info_folder(self):
18         folder = os.path.join(self.data_folder, "DataInfo")
19         if not os.path.isdir(folder):
20             os.makedirs(folder)
21         return folder
22
23     # 定义数据信息文件所在路径的 property
24     @property
25     def data_info_file(self):
26         return os.path.join(self.data_info_folder, "{}.info".format(self.name))
27
28     # 定义训练数据文件所在路径的 property
29     @property
30     def train_data_file(self):
31         return os.path.join(self.data_cache_folder, "train.npy")
32
33     # 定义测试数据文件所在路径的 property
34     @property
35     def test_data_file(self):
36         return os.path.join(self.data_cache_folder, "test.npy")
```